CODE4STRUCT: Code Generation for Few-Shot Structured Prediction from Natural Language

Xingyao Wang and **Sha Li** and **Heng Ji**
University of Illinois at Urbana-Champaign, IL, USA
{xingyao6, shal2, hengji}@illinois.edu

Abstract

Large Language Model (LLM) trained on the mixture of text and code has demonstrated impressive capability in translating natural language (NL) into structured code. In this work, we propose CODE4STRUCT to leverage such text-to-structure translation capability to tackle structured prediction tasks in NLP. For example, Event Argument Extraction (EAE) aims to convert text into event-argument structures that can be represented as a class object using code. This alignment between structures and code enables us to take advantage of Programming Language (PL) features such as inheritance[1] and type annotation[2] to introduce external knowledge or add constraints with ease. We exploit the analogy between PL and NLP problems, and, as a case study, we use CODE4STRUCT to tackle the EAE task using code generation. We ask a LLM to generate code to instantiate an event class with predicted arguments given a NL sentence. Despite only using 50 training instances for each event type, CODE4STRUCT is comparable to fully-supervised models trained on 4,202 event instances and, when given the same 50-shot data, outperforms current state-of-the-art (SOTA) by 20.8% absolute F1. When prompted with hierarchical event types implemented using inheritance, CODE4STRUCT can predict arguments for low-resource event types using 10-shot training instances from its sibling event type and outperforms zero-shot baseline by 12% absolute F1.[3]

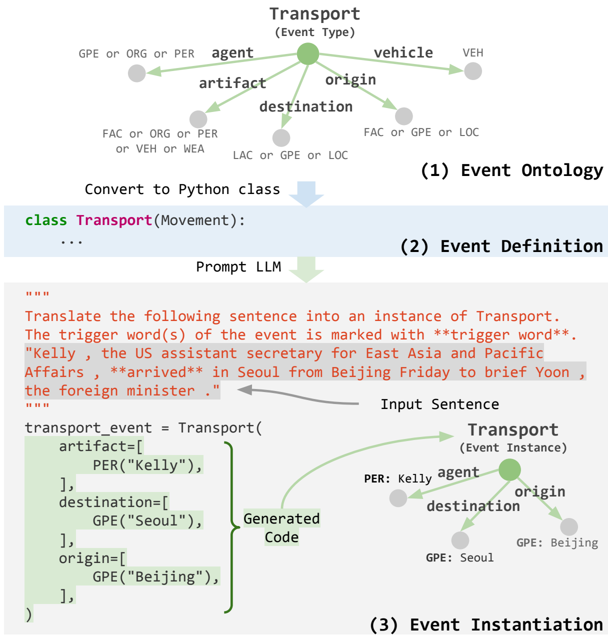

Figure 1: Event Argument Extraction using code generation. We convert the existing event type ontology to PYTHON class definitions. Conditioned on these definitions, we put the input sentence for event extraction into a docstring as the prompt for code generation. The generated code (colored in green) can be mapped to an instance graph of `Transport` event.

1 Introduction

Large Language Model (LLM) trained on massive corpora of code mixed with natural language (NL) comments and docstrings[4] (*e.g.*, Chen et al. 2021, Nijkamp et al. 2022) has demonstrated the ability to translate natural language instructions into code that can solve 75% of the `HumanEval` coding exercises (Chen et al., 2021) specified in NL. We ask if this conversion between language and code can serve as a bridge to build a connection between language and semantic structure, which is the goal of many structured prediction tasks (*e.g.*, semantic parsing, information extraction) in Natural Language Processing (NLP). In particular, the target structure (*e.g.*, event-entity graph connected by argument edges in Figure 1) can be mapped to code in a more straightforward way compared

[1] Inheritance is a way to create a hierarchy of classes in PL. A child class can base upon another class, retaining similar implementation.

[2] Developers use type annotations to indicate the data types of variables and input/outputs of functions.

[3] All code and resources will be made publicly available at https://github.com/xingyaoww/code4struct.

[4] Texts used to document a specific segment of code.

Event Argument Extraction	**Programming Language (Python)**
Event / Entity Type `Transport, VEH`	**Class definition** `class Transport, class VEH`
Hierarchical Events `Movement:Transport`	**Inheritance** Inheritance is a way to create a hierarchy of classes in PL. A child class can base upon another class, retaining similar implementation. `class Transport(Movement)`
Event Arguments `vehicle`	**Function arguments** `def function(vehicle=...)`

Argument Constraint
Each argument can has **multiple** entities; Argument *vehicle* should be entities of type **VEH**.
	Type Annotation
	Type annotations are used by developers to indicate the data types of variables and input/outputs of functions.
	`def function(`
	`    vehicle: List[VEH] = [], ...`
	`)`

Weakly-supervised Information
Transport Event describes *someone* transporting *something* in a *vehicle* from *one place* to *another place*.
	Docstring or Comments
	`class Transport(Movement):`
	`    """`
	`    self.agent transported self.artifact in self.vehicle vehicle from self.origin place to self.destination place.`
	`    """`

Table 1: Mapping between Event Argument Extraction requirements and features of Python programming language.

to natural language, which often requires careful handcrafting (Hsu et al. 2022, Li et al. 2021, Table 2). This alignment between structures and code allows us to better utilize LLM, compared to language generation, to generate structures through code generation. In addition, programming languages have an inherent advantage in representing complex and interconnected structures (Miller, 1981) with features such as inheritance and type annotation. As a case study, we showcase our proposed CODE4STRUCT on the Event Argument Extraction (EAE) task, which aims to extract event structures from unstructured text. EAE is the ideal testbed for our method due to the close alignment between EAE and PL as shown in Table 1. In CODE4STRUCT (Figure 1), we first translate the entity and event type ontology into Python class definitions. Conditioned on relevant class definitions and the input sentence, we use CODEX (Chen et al., 2021) to generate an instantiation of the event class, from which we can extract the predicted arguments. By leveraging the alignment between PL and NLP problems like EAE, CODE4STRUCT enjoys various advantages. First, we can naturally enforce argument constraints (*i.e.*, entity types accepted by each argument) for output structures using PL features like type annotation. Second, we can naturally utilize the event hierarchy by leveraging inheritance (Table 1). Inheritance allows a child event class to reuse most components of its parent class while preserving its unique property. We demonstrate that hierarchical event types allow low-resource event types to use annotated training examples from their high-resource siblings (§4.4). Third, the flexibility of our formulation allows for performing EAE and Entity Classification (*i.e.*, predicting the entity type of extracted arguments) for the extracted arguments simultaneously. Finally, our formulation handles zero or multiple argument fillers for the same argument role with ease by annotating the expected type and default value for each argument to be a Python list.

We outline our contributions as follows:

- We propose CODE4STRUCT to tackle structured prediction problems in NLP using code generation with LLM trained on a mixture of language and code.

- As a case study, we use CODE4STRUCT to tackle Event Argument Extraction (EAE). We demonstrate that 50-shot CODE4STRUCT rivals fully-supervised methods trained on 4,202 instances. CODE4STRUCT outperforms a SOTA approach by 20.8% absolute F1 gain when 50-shot data are given to both (§4.2).

- We show how a number of requirements and extensions for EAE can be conveniently mapped to code, including type constraints and event type hierarchy.

- We demonstrate that event hierarchy implemented by inheritance can improve prediction, compared to zero-shot baseline, by 12% F1 for low-resource event types using 10-shot examples from their sibling event types (§4.4).

2 Code Generation Prompt Construction

We prompt a LLM (*i.e.*, CODEX) with the definitions of event types and argument roles from an ontology and input sentences to generate code that instantiates the given event type. We breakdown the input prompt into three components: (1) ontology code which consists of Python class definitions for entity types and event types (§2.1); (2) optional k-shot in-context learning examples for the aforementioned event type (§2.3); (3) task prompt for completion (§2.2). We show a breakdown of the full prompt in Figure 2.

2.1 Ontology Code Context

A code context is a concatenation of the base class definition, entity class definitions, and event class definitions.

Base Class Definition We define base type `Entity` and `Event` to be inherited by other classes for distinction.

Entity Class Definition We use entity type definitions from the Automatic Content Extraction (ACE) program (Doddington et al., 2004). We construct Python classes that inherit from `Entity` and use the entity type as class name (*e.g.*, `class GPE(Entity)`). We add a natural language description as a docstring of the defined Class for each entity type.

Event Class Definition We define the event class using the name of the event type (*e.g.*, `class Transport`). Mimicking class definitions in Object-Oriented PL, we inherit the event class definition from its parent (*e.g.*, `class Transport(Movement)`) or root event type if the event class does not has a parent (*e.g.*, `class Movement(Event)`). We only include class definition for child event type (*e.g.*, `class Transport`) in CODE4STRUCT, except in §4.4. We define the argument roles (*e.g.*, destination of `Transport`) as input arguments of the constructor function `__init__`. We specify the type of each argument role using Python type annotation, a commonly used PL feature: For example, `agent: List[GPE | ORG | PER]` means that the `agent` argument accepts a list of entities such as GPE (Geo-Political Entity), ORG (Organization), or PER (Person). We assign each input argument (*e.g.*, `agent`) to a class member variable of the same name following a common pattern in PL. We include event description templates into a docstring of the class definition. We modify event description templates from Li et al. (2021) by replacing each role with their corresponding member variable (*e.g.*, `self.agent`).

2.2 Task Prompt

The task prompt consists of a docstring describing the task and an incomplete event instantiation code for completion. An example of a task prompt can be found in Figure 2. The text-based docstring contains a task instruction and an input sentence. We mark the ground truth trigger words for the input text by surrounding them with `**`. We choose to use `**` as it is used to set text to bold in `Markdown` (a markup language for creating formatted text), which is commonly found in code bases and web data on which our LLM is trained. The incomplete code prompt assigns a partial instantiation of an event class to a variable to trigger the model for completion, for example, `transport_event = Transport(`.

We observed that CODEX tends to generate more novel sentences paired with extracted arguments if no stopping constraint is applied. To focus on the EAE task, we stop the code generation whenever any of the pattern `"""`, `class`, `print`, or `#` is generated by the model. We note that this capability to generate novel content (*e.g.*, class definitions for new event types, new sentences paired with extracted arguments) has the potential to discover new event types and generate new training examples to enhance the model's EAE performance.

2.3 In-context Learning

Optionally, we can include in-context learning examples which are task prompts (§2.2) paired with completed event instantiations using ground-truth arguments (see Figure 2 for a specific example). For k-shot learning, we concatenate k such examples together. Given a task prompt, we gather in-context learning examples by collecting training instances with the same event type, following the order of occurrences in the training set.

3 Why Represent Event Structure in PL?

A wide range of NLP tasks have benefited from LLM (Brown et al., 2020; Hoffmann et al., 2022;

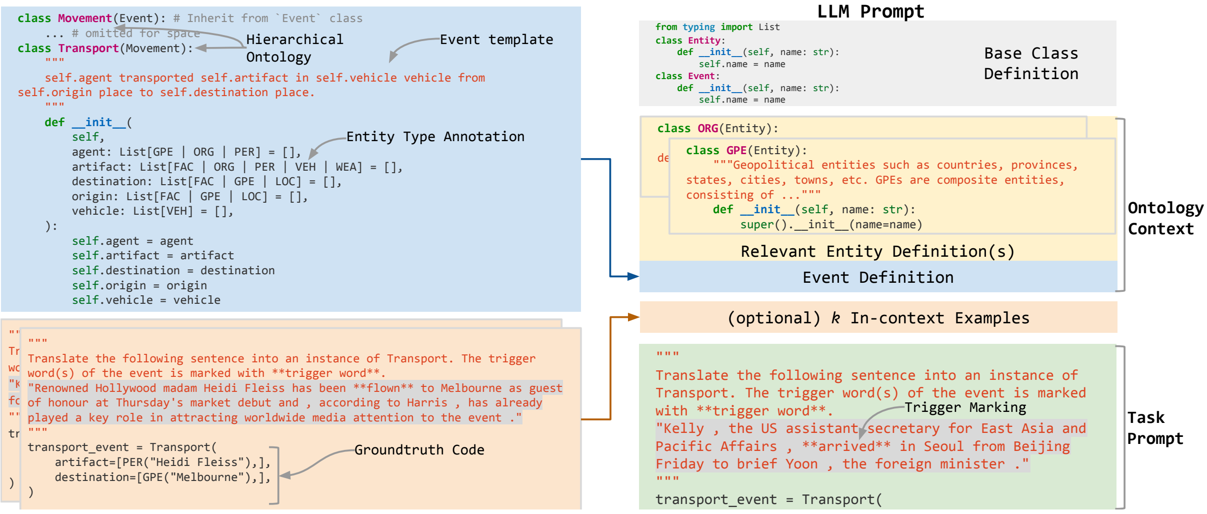

Figure 2: Prompt components. **(1)** Ontology code context contains definitions of entity and event classes, colored in yellow and blue (§2.1). **(2)** K-shot examples for in-context learning, colored in orange (§2.3). **(3)** The task prompt, appended at the end with partial class instantiation for LLM completion, colored in green (§2.2).

Prior Work	Language Template
DEGREE (Hsu et al., 2022)	**somebody** was moved to **somewhere** from **some place** by **some way**. **somebody** or **some organization** was responsible for the movement. **something** was sent to **somewhere** from **some place**. **somebody** or **some organization** was responsible for the **transport**.
BART-Gen (Li et al., 2021)	**<arg1> transported <arg2>** in **<arg3>** vehicle from **<arg4>** place to **<arg5>** place
Text2Event (Lu et al., 2021)	((**Transport** returned (Agent **<arg>**) (Artifact **<arg>**) (Destination **<arg>**) (Origin **<arg>**) (Vehicle **<arg>**))

Table 2: Example of language templates for Event Argument Extraction used by Hsu et al. (2022); Li et al. (2021); Lu et al. (2021).

Chowdhery et al., 2022) trained on web-scale language corpora. To effectively use LLM trained on language for EAE, one of the biggest challenges is to specify the desired output, namely event structures, using natural language.

There is a tradeoff between the effort put into defining the output or designing the prompt (*e.g.*, Text2Event in Table 2) and the benefit from pretraining in natural language (*e.g.*, DEGREE and BART-Gen in Table 2). Text2Event (Lu et al., 2021) resides at one end of the spectrum with a concise but unnatural output format. As a result, this formulation under-utilizes the pretraining power of the model and does not work in low-resource settings. Towards the other end, Hsu et al. (2022); Li et al. (2021) design manual templates for the model to fill in. We also design a natural language prompt as shown in Figure A.1 that closely matches the semantics of our code prompt for comparison. Note that the natural language prompt is much more verbose and as we show in §4.3, it results in sub-optimal performance.

Essentially, this tradeoff is a consequence of the mismatch between the pretraining corpora format and the task output format. Instead of using LLM trained on only unstructured text, we turn to LLM trained with a mixture of text and code, where the text is often aligned in semantics to the accompanying code. Such Code-LLMs have the ability to convert text into corresponding code as demonstrated by (Chen et al., 2021; Nijkamp et al., 2022). Then we can map the desired output event structure into code in a straightforward manner and leverage the full pretraining power of these models. PLs like Python offer features (*e.g.*, class, docstrings, type annotations, inheritance) that have a significant presence in the pre-training corpus of Code-LLM due to frequent usage. CODE4STRUCT leverages these features to succinctly describe event structures, which makes it better aligned with Code-LLM. By leveraging LLM's learned knowledge from diverse pre-training domains, CODE4STRUCT can work well in open-domain, achieving non-trivial zero-shot performance given unseen event types (§4.2). CODE4STRUCT is also data-efficient as exemplified by reaching comparable performance to fully-supervised methods with much fewer annotated examples (1% per event type) (§4.2).

4 Experiments

4.1 Experiment Setup

LLM We use CODEX[5] (Chen et al., 2021), a 12B GPT-3 model finetuned on publicly available code from Github. We access CODEX through OpenAI API, which supports an input length of up to 8k tokens. We also compare its performance with GPT-3[6] (Brown et al., 2020) through OpenAI API which supports up to 4k tokens for input.

Hyperparameters We prompt CODEX to generate code that instantiates an event using sampling temperature $t = 0$ (*i.e.*, greedy decoding). We set the max number of new tokens for each generation to 128, which fits all code outputs for the test set.

Evaluation Tasks We use ground truth event type and gold-standard trigger words to perform EAE. We also evaluate our model's performance on Entity Classification for extracted arguments since our approach natively supports it.

Dataset We evaluate our performance of EAE and Entity Classification on the English subset of Automatic Content Extraction 2005 dataset (ACE05-E)[7] (Doddington et al., 2004). We follow Wadden et al. (2019); Lin et al. (2020) for dataset processing. ACE05-E has hierarchical event types with 8 parent types and 33 child types. Among all child types, roughly half of the event types (14 out of 33) in ACE05-E have less than 50 event instances in the training set. We show statistics for each event type in Table A.2.

Evaluation metrics We use **Argument F1-score** following prior work (Li et al., 2021; Hsu et al., 2022): We consider an argument to be correctly identified when the head word of the predicted text span matches that of the human-annotated span (denoted as **Arg-I**); We consider an argument to be correctly classified if the role (*e.g.*, `agent`) of an *correctly identified* argument matches that of the human annotation (denoted as **Arg-C**); Unlike most prior work, we also consider whether the predicted entity type (*e.g.*, GPE) matches the ground truth for a *correctly identified and classified* argument (denoted as **Arg-C+E**).

[5] `code-davinci-002`

[6] `text-davinci-002`

[7] https://www.ldc.upenn.edu/collaborations/past-projects/ace

Baselines Unlike prior methods trained on the entire training set, CODE4STRUCT learns from up to 50 examples (*i.e.*, 39 examples per event type on average, roughly 1% among all training instances) to predict arguments for each test event type. To ensure a fair comparison, for each event type t in the test set, we train a DEGREE model (Hsu et al., 2022) on 50-shot in-context examples CODE4STRUCT used. We evaluate the DEGREE model trained on event type t on a partition of the test set that only contains instances of event type t. We then aggregate F1 scores (micro F1) across all 31 event types on the test set and report them in Table 3 (DEGREE 50-shot). Following Hsu et al. (2022), we also compare with classification-based (DyGIE++, BERT_QA, OneIE) or generation-based (TANL, BART-Gen, DEGREE) models trained on the full training set.

4.2 Comparison with Supervised Models

We report the performance of CODE4STRUCT in comparison with prior work in Table 3 and examine our model's performance with varying number of examples in Table 4.

In the few-shot setting, CODE4STRUCT can surpass DEGREE, the current state-of-the-art, by a large margin (20.8% absolute F1 difference on Arg-C). Our zero-shot CODE4STRUCT model can already achieve higher Arg-I performance than the 50-shot DEGREE. Despite only using 39 examples per event type on average, our approach achieves comparable performance with other fully-

Model	Data	Arg-I F1	Arg-C F1
DyGIE++	Full	66.2	60.7
BERT-QA	Full	68.2	65.4
OneIE	Full	73.2	69.3
TANL	Full	65.9	61.0
BART-Gen	Full	69.9	66.7
DEGREE	Full	**76.0**	**73.5**
DEGREE	50-shot*	40.8	37.3
CODE4STRUCT	50-shot*	**62.0**	**58.1**
CODE4STRUCT	0-shot	50.0	35.7

Table 3: Performance (in %) comparison with existing approaches with ground truth triggers on ACE05-E. *Some rare event types do not have 50 examples present in the training set, on average we have 39 examples per event type. Our approach achieves comparable performance despite only using up to 50 examples per event. We report the performance of supervised models using full dataset from Hsu et al. (2022).

k-shot	Arg-I F1	Arg-C F1	Arg-C+E F1
0	50.0	35.7	23.1
1	58.3	49.1	39.4
5	58.2	53.3	48.3
10	57.8	53.5	48.4
20	61.2	57.1	51.1
50	**62.0**	**58.1**	**54.0**

Table 4: Few-shot performance (in %) measured by F1 score. We observe improvements with diminishing returns when we increase the number of in-context examples.

supervised models trained on 100% of the training data (4,202 instances).

We observe that F1 scores for all tasks increase when providing more in-context learning examples, but with diminishing returns. The initial in-context example ($k = 1$) brings the largest absolute performance gain (about 13 F1 in Arg-C). We stop at 50 in-context examples ($k = 50$) as including more examples would exceed the input length limitation imposed by CODEX.

4.3 Comparison with Text Prompt

To compare our code-based prompt with text-style GPT-3 prompt, we design a text prompt mimicking our code prompt in Figure A.1. It has similar components as our code-based prompt in Figure 2. The text prompt relies on natural language to define the requirement and format of the desired output, while the code prompt utilizes PL syntax.

We compare the performance of text prompt and code prompt (§2) on 175B GPT-3 (Brown et al., 2020) and 12B CODEX (Chen et al., 2021) in Figure 3. We also tested the code prompt on GPT-3 as its pre-training corpus contains a small fraction of code.

We summarize our findings as follows:

- CODEX + code prompt outperforms text prompt on all metrics under the few-shot setting, despite CODEX being 14 times smaller than GPT-3. The performance gap is most significant on Arg-C F1 (8.7% absolute F1 difference when compared to GPT-3 + text prompt).
- Zero-shot code prompt under-performs text prompt for both CODEX and GPT-3 on Arg-C and Arg-C+E. Evidence in §A.1 suggests that prompt design plays a more important role in determining zero-shot performance.

  We speculate this superior zero-shot performance could be attributed to text prompt's ability to elicit learned event related-information (*e.g.*, event keywords, hierarchy) from LLM.
- GPT-3 underperforms CODEX when using the same code prompt in the low-shot regime. However, when more in-context examples are given, GPT-3 + code prompt performs on par with CODEX on Arg-I, while being consistently worse on other metrics.
- Interestingly, we observe a much smaller gap between code prompt and text prompt on identifying entity spans (Arg-I) compared to classifying them (Arg-C) for both GPT-3 and CODEX. That is, the text prompt is able to successfully extract the entity and recognize its type (*e.g.*, `GPE("Seoul")`) yet often

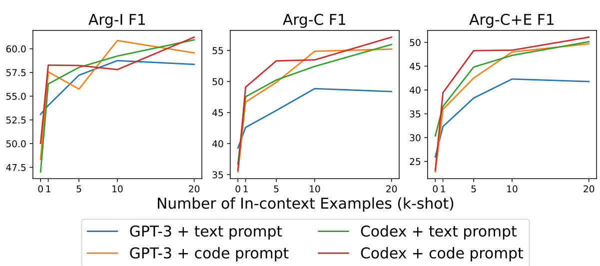

Figure 3: Performance (in %) comparison between text-based prompt and code-based prompt on different models. Argument extraction performance (Arg-C F1) of code-based prompt on CODEX (red line) surpasses the performance of text-based prompt on GPT-3 with an observable margin under the few-shot setting.

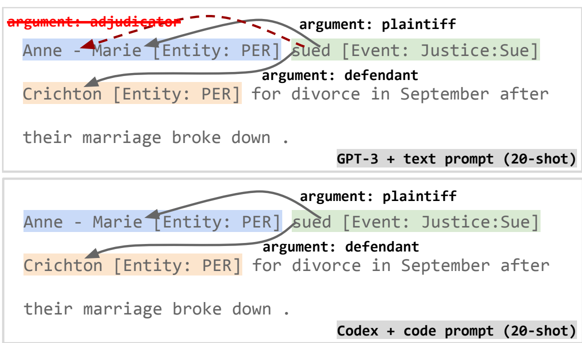

Figure 4: Example prediction of 20-shot GPT-3 + text prompt and CODEX + code prompt. In this example, GPT-3 + text prompt incorrectly predicts the same entity *Anne-Marie* as both *adjudicator* and *plaintiff* of the `Justice:Sue` event.

fails to link it to the corresponding role (*e.g.*, `destination=GPE("Seoul")`). We show a similar qualitative example in Figure 4. This shows that code is a better medium for representing relations between the event and arguments. Pretraining on code can also grant the model the ability to handle relations betters, as shown by the good performance of CODEX + text prompt compared to GPT-3 + text prompt.

4.4 Event Type Hierarchy Improves Low-resource EAE

In this section, we show that CODE4STRUCT, when provided with hierarchical event definitions and few-shot training instances from a sibling event type (*e.g.*, `Transfer_Money`) under the same parent event type (*e.g.*, `Transaction`), can improve performance for child event types (*e.g.*, `Transfer_Ownership`) as good as if training instances from the same event type were used. This allows low-resource event types that have no annotated data to exploit the event type hierarchy and benefit from their high-resource siblings. We include an example task prompt with sibling examples in Figure A.6 and report our results in Table 5.

Setup We split the child types for each parent type into training and testing types by selecting the high-resource child type with the largest amount of training instances to be the training type and have the rest be testing types. The train-test split for ACE types can be found in Table A.3. Under the same parent event type, we use data instances from the training type (*i.e.*, a sibling of testing types) as in-context examples to predict arguments for each testing type. We include event class definition (Figure 2) for parent event type (*e.g.*, `Transaction`), child training (sibling) event type (*e.g.*, `Transfer_Money`), and child testing event type (*e.g.*, `Transfer_Ownership`).

	Arg-I F1	Arg-C F1	Arg-C+E F1
0-shot	52.8	42.9	29.9
1-shot (same type)	54.3	50.2	**44.9**
1-shot (sibling type)	**57.2**	**51.9**	43.8
10-shot (same type)	58.7	**55.2**	**50.0**
10-shot (sibling type)	**60.8**	54.9	49.8

Table 5: Model performance (in %) when using hierarchical ontology and sibling examples for in-context learning. To ensure a fair comparison, F1 scores are aggregated from 23 test event types in Table A.3 that contains more than 10 training instances.

We show an example of event definition with sibling type in Figure A.5. The few-shot performance when using data from sibling type is denoted with *(sibling type)* in Table 5. To demonstrate the effectiveness of using data from sibling event types, we compare it with using training instances from the testing event type itself for in-context learning, which is denoted as *(same type)* in Table 5.

Results We observe that CODE4STRUCT, when prompted with training examples from sibling type, performs on par with the prompt that uses training examples from the testing type itself on 1-shot and 10-shot. The strong performance gain (9% Arg-C F1 on 1-shot, 12% Arg-C F1 on 10-shot, compared with 0-shot) contributed by training examples from sibling type demonstrate the potential of applying CODE4STRUCT to low-resource event types with no training data by exploiting their hierarchical relationship with other high-resource event types.

5 Qualitative Analysis

We show examples of 0-shot and 50-shot CODE4STRUCT argument extraction result in Figure 5. CODE4STRUCT can leverage implicit commonsense knowledge in LLM to infer arguments not presented in the text. In the first 0-shot example, the model inferred the `place` of Welch's retirement is in the *United States*. This is a reasonable guess since Welch, in this example, is the former CEO of General Electric (GE), whose headquarter is in the United States. In the second 0-shot example, our model inferred that the `Justice:Fine` event should take `place` in a *court*, which matches our commonsense knowledge. Interestingly, we observe that increasing the number of in-context examples from 0-shot to 50-shot inhibits LLM from generating arguments, including these inferred arguments and a correctly predicted argument (*i.e.*, `SEC`) in 0-shot predictions.

6 Related Work

Large Language Model (LLM) LLM (Brown et al., 2020; Hoffmann et al., 2022; Chowdhery et al., 2022) trained on large-scale web corpora has shown emergent capabilities to perform few-shot in-context learning and has demonstrated strong performance on many downstream NLP tasks (*e.g.*, question answering, cloze, commonsense reasoning, natural language inference, *etc.*). Following the success of LLM in NLP tasks, Chen et al. (2021); Nijkamp et al. (2022) trained LLM on both

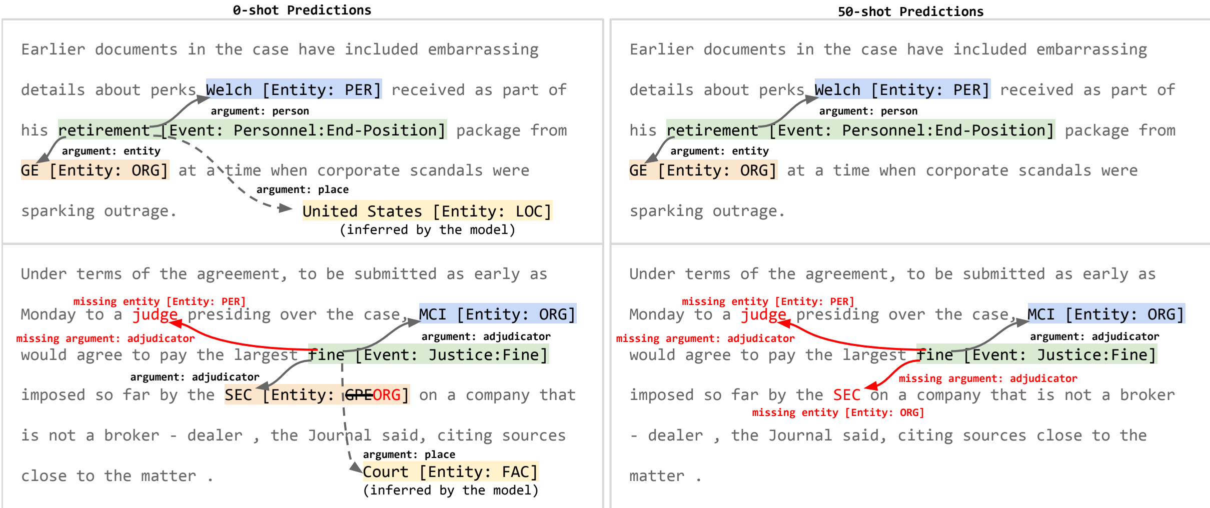

Figure 5: Examples of 0-shot and 50-shot CODE4STRUCT event argument prediction on ACE05-E. In both 0-shot examples, LLM can infer an entity that does not present in the text as an argument (marked with a yellow span). CODE4STRUCT predicts fewer arguments when the examples are increased to 50-shot. We mark incorrect predictions with strikethrough text. Entities that LLM failed to predict are marked in red font.

natural language and programming language and demonstrated surprising code generation capabilities, which led to production applications (Copilot).

Event Extraction Li et al. (2013); Nguyen et al. (2016); Yang and Mitchell (2016); Wadden et al. (2019); Lin et al. (2020) use classification models and mitigate error propagation from pipeline models by leveraging global features to jointly predict event triggers and arguments. Recent work such as Liu et al. (2020) formulates event extraction as a reading comprehension problem and Li et al. (2021); Huang et al. (2021); Paolini et al. (2021); Hsu et al. (2022) converts event extraction to a text generation task to better exploit label semantics from pretrained language models. The most similar work to ours is Text2Event (Lu et al., 2021), which uses controlled generation to directly generate structures in manually specified linearized format which hinders the model in leveraging pre-trained NL knowledge. On the other hand, our approach CODE4STRUCT directly generates structure in PL instead of using a manually designed format to fully exploit LLM's knowledge of PL.

Code-LLM for Structured Task Sun et al. (2019); Singh et al. (2022) focus on procedural tasks that aim to control situated agents in an embodied environment by representing the procedure plan in code. Madaan et al. (2022) uses Code-LLM to generate a structured commonsense reasoning graph represented in code, which is similar in spirit to our work but in a different task. We also leverage PL features like inheritance to exploit hierarchical information for structured prediction which is largely overlooked by prior work.

7 Conclusions and Future Work

In this paper, we propose CODE4STRUCT to tackle structured prediction tasks in NLP by leveraging LLM trained on both natural and programming languages. As a case study, we use CODE4STRUCT to extract event arguments from natural language sentences through code generation. Our proposed CODE4STRUCT rivals fully-supervised models trained on 4,202 data instances, despite only using 50 training instances for each event type. It also outperforms a SOTA model by 20.8% absolute F1 when both are given the same 50-shot data. Furthermore, benefit from hierarchical event definitions, CODE4STRUCT can predict arguments for low-resource event types only using 10-shot training instances from its sibling event type and outperforms 0-shot baseline by 12% absolute F1 score. Going forward, we plan to expand CODE4STRUCT to a broader range of structured prediction tasks (*e.g.*, EAE with time arguments, relation prediction) and support multilingual input. We would further explore the alignment between NL and PL to predict future events by reasoning through existing event structures represented using code.

References

Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. 2013. Abstract Meaning Representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, pages 178–186, Sofia, Bulgaria. Association for Computational Linguistics.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*.

Github Copilot. https://github.com/features/copilot.

George Doddington, Alexis Mitchell, Mark Przybocki, Lance Ramshaw, Stephanie Strassel, and Ralph Weischedel. 2004. The automatic content extraction (ACE) program – tasks, data, and evaluation. In *Proceedings of the Fourth International Conference on Language Resources and Evaluation (LREC'04)*, Lisbon, Portugal. European Language Resources Association (ELRA).

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. 2022. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*.

I-Hung Hsu, Kuan-Hao Huang, Elizabeth Boschee, Scott Miller, Prem Natarajan, Kai-Wei Chang, and Nanyun Peng. 2022. DEGREE: A data-efficient generation-based event extraction model. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1890–1908, Seattle, United States. Association for Computational Linguistics.

Kung-Hsiang Huang, Sam Tang, and Nanyun Peng. 2021. Document-level entity-based extraction as template generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 5257–5269, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Qi Li, Heng Ji, and Liang Huang. 2013. Joint event extraction via structured prediction with global features. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 73–82, Sofia, Bulgaria. Association for Computational Linguistics.

Sha Li, Heng Ji, and Jiawei Han. 2021. Document-level event argument extraction by conditional generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 894–908, Online. Association for Computational Linguistics.

Ying Lin, Heng Ji, Fei Huang, and Lingfei Wu. 2020. A joint neural model for information extraction with global features. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7999–8009, Online. Association for Computational Linguistics.

Jian Liu, Yubo Chen, Kang Liu, Wei Bi, and Xiaojiang Liu. 2020. Event extraction as machine reading comprehension. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1641–1651, Online. Association for Computational Linguistics.

Yaojie Lu, Hongyu Lin, Jin Xu, Xianpei Han, Jialong Tang, Annan Li, Le Sun, Meng Liao, and Shaoyi Chen. 2021. Text2Event: Controllable sequence-to-structure generation for end-to-end event extraction. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 2795–2806, Online. Association for Computational Linguistics.

Aman Madaan, Shuyan Zhou, Uri Alon, Yiming Yang, and Graham Neubig. 2022. Language models of code are few-shot commonsense learners. *arXiv preprint arXiv:2210.07128*.

L. A. Miller. 1981. Natural language programming: Styles, strategies, and contrasts. *IBM Systems Journal*, 20(2):184–215.

Thien Huu Nguyen, Kyunghyun Cho, and Ralph Grishman. 2016. Joint event extraction via recurrent neural networks. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 300–309, San Diego, California. Association for Computational Linguistics.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A conversational paradigm for program synthesis. *arXiv preprint*.

Giovanni Paolini, Ben Athiwaratkun, Jason Krone, Jie Ma, Alessandro Achille, Rishita Anubhai, Cicero Nogueira dos Santos, Bing Xiang, and Stefano Soatto. 2021. Structured prediction as translation between augmented natural languages. *arXiv preprint arXiv:2101.05779*.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. 2022. ProgPrompt: Generating situated robot task plans using large language models.

Shao-Hua Sun, Te-Lin Wu, and Joseph J Lim. 2019. Program guided agent. In *International Conference on Learning Representations*.

David Wadden, Ulme Wennberg, Yi Luan, and Hannaneh Hajishirzi. 2019. Entity, relation, and event extraction with contextualized span representations. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5784–5789, Hong Kong, China. Association for Computational Linguistics.

Bishan Yang and Tom M. Mitchell. 2016. Joint extraction of events and entities within a document context. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 289–299, San Diego, California. Association for Computational Linguistics.

Zixuan Zhang and Heng Ji. 2021. Abstract Meaning Representation guided graph encoding and decoding for joint information extraction. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 39–49, Online. Association for Computational Linguistics.

A Appendix

A.1 Prompt Component Analysis

In this section, we present an empirical analysis of other prompt component candidates. We compare different prompt components in Table A.1 following the same set of hyper-parameters described in §4.1.

- **Event Keywords** We augment event-related keywords into the docstring of event definition for CODE4STRUCT (illustrated in Figure A.3). We follow the same keywords used by Li et al. (2021).
- **Event Hierarchy** We experiment with hierarchical event definitions by including parent class definition (*e.g.*, `Movement`) into the prompt. An example of hierarchical event definition can be found in Figure A.4.
- **AMR** Zhang and Ji (2021) have demonstrated the effectiveness of utilizing Abstract Meaning Representation (AMR) (Banarescu et al., 2013) for information extraction. We experiment with AMR-augmented prompts. We use `armlib` [8] to predict AMR, and append the AMR structure after the NL sentence in the task prompt §2.2 (see Figure A.2 for an example).

Prompts that include event keywords, hierarchy, and AMR all perform better than CODE4STRUCT under the zero-shot setting on all metrics (Table A.1). The AMR-augmented prompt is effective and outperforms CODE4STRUCT and all other designs under zero-shot by a relatively large margin (1.8 absolute F1 on Arg-C compare to CODE4STRUCT). CODE4STRUCT + hierarchy slightly outperforms CODE4STRUCT for 20-shot and 50-shot.

[8] https://github.com/bjascob/amrlib, parse_xfm_bart_large v0.1.0

k_shot	Arg-I F1 0	1	10	20	50	Arg-C F1 0	1	10	20	50	Arg-C+E F1 0	1	10	20	50
CODE4STRUCT	50.0	**58.3**	57.8	61.2	62.0	35.7	**49.1**	53.5	57.1	**58.1**	23.1	**39.4**	**48.4**	51.1	54.0
+ amr	50.6	55.3	56.6	-	-	**37.5**	45.0	51.8	-	-	**23.5**	37.8	47.1	-	-
+ hierarchy	50.6	57.3	57.2	**62.1**	**62.3**	36.0	47.8	52.8	**58.5**	**58.1**	23.2	38.3	47.0	**52.1**	**54.1**
+ keywords	**52.4**	58.2	**58.7**	62.0	61.8	35.6	48.7	**53.6**	58.1	57.9	21.2	39.0	47.5	51.3	53.6

Table A.1: Prompt components analysis. The best scores (in %) are bolded. - means the result is unavailable due to the input prompt exceeding CODEX supported input token length.

```
Description of base entity types:
GPE: Geopolitical entities such as countries, provinces, states, cities, towns,
etc. GPEs are composite entities, consisting of a physical location, a government,
and a population. All three of these elements must be present for an entity to be
tagged as a GPE. A GPE entity may be a single geopolitical entity or a group.
... (other types omitted for space)
                                                    (1) Entity Definition(s)
```

```
Role definition of event type Transport (Parent type: Movement):
1. agent (need to be one of GPE or ORG or PER)
2. artifact (need to be one of FAC or ORG or PER or VEH or WEA)
3. destination (need to be one of FAC or GPE or LOC)
4. origin (need to be one of FAC or GPE or LOC)
5. vehicle (need to be one of VEH)
Multiple entities can be extracted for the same role, each entity is a
double-quote enclosed string.
Each extracted entity should look like: (Base Entity Type) "content of extracted
string"
If entity is not present in the text, write: () ""
Different entities are delimited by a comma.
In this event: [agent] transported [artifact] in [vehicle] vehicle from [origin]
place to [destination] place.
                                                    (2) Event Definition
```

```
Translate the following sentence into an instance of Transport event. The
trigg
"Kel
Sunda
1. ag
2. ar
3. de
4. or
5. ve
```

```
Translate the following sentence into an instance of Transport event. The
trigger word(s) of the event is marked with **trigger word**.
"Renowned Hollywood madam Heidi Fleiss has been **flown** to Melbourne as
guest of honour at Thursday's market debut and , according to Harris , has
already played a key role in attracting worldwide media attention to the
event ."
1. agent: () ""
2. artifact: (PER) "Heidi Fleiss"
3. destination: (GPE) "Melbourne"
4. origin: () ""
5. vehicle: () ""
                                                 (3) k In-context Examples
```

```
Translate the following sentence into an instance of Transport event. The trigger
word(s) of the event is marked with **trigger word**.
"Kelly , the US assistant secretary for East Asia and Pacific Affairs ,
**arrived** in Seoul from Beijing Friday to brief Yoon , the foreign minister ."
1. agent: () ""
2. artifact: (PER) "Kelly"
3. destination: (GPE) "Seoul"
4. origin: (GPE) "Beijing"
5. vehicle: () ""
                                                    (4) Event Instantiation
```

Figure A.1: Natural language prompt for EAE task. We ask GPT-3 to generate event instantiation marked in green.

```
"""
Translate the following sentence into an instance of Transport. The trigger
word(s) of the event is marked with **trigger word**.
"Kelly , the US assistant secretary for East Asia and Pacific Affairs ,
**arrived** in Seoul from Beijing Friday to brief Yoon , the foreign minister ."

Abstract Meaning Representation of the given sentence:
(a / arrive-01
      :ARG1 (p / person
            :name (n / name
                  :op1 "Kelly")
            :ARG0-of (h / have-org-role-91
                  :ARG1 (g / government-organization
                        :name (n2 / name
                              :op1 "East"
                              :op2 "Asia"
                              :op3 "and"
                              :op4 "Pacific"
                              :op5 "Affairs")
                        :poss (c / country
                              :name (n3 / name
                                    :op1 "US")))
                  :ARG2 (s / secretary
                        :mod (a2 / assistant))))
      :ARG3 (c2 / city
            :name (n4 / name
                  :op1 "Beijing"))
      :ARG4 (c3 / city
            :name (n5 / name
                  :op1 "Seoul"))
      :time (d / date-entity
            :weekday (f / friday))
      :purpose (b / brief-01
            :ARG0 p
            :ARG1 (p2 / person
                  :name (n6 / name
                        :op1 "Yoon")
                  :ARG0-of (h2 / have-org-role-91
                        :ARG2 (m / minister
                              :topic (f2 / foreign))))))
"""
transport_event = Transport(
```

Figure A.2: Example of an AMR-augmented task prompt. We append the AMR prediction after the input sentence. Different prompt components compared to CODE4STRUCT are highlighted in yellow.

```
class Transport(Movement):
    """
    self.agent transported self.artifact in self.vehicle vehicle from self.origin place to self.destination place.
    Event keywords: transport, move, travel, head.      Event Keywords
    """
    def __init__(
        self,
        agent: List[GPE | ORG | PER] = [],
        artifact: List[FAC | ORG | PER | VEH | WEA] = [],
        destination: List[FAC | GPE | LOC] = [],
        origin: List[FAC | GPE | LOC] = [],
        vehicle: List[VEH] = [],
    ):
        self.agent = agent
        self.artifact = artifact
        self.destination = destination
        self.origin = origin
        self.vehicle = vehicle
```

Figure A.3: Example of an event-keywords-augmented event definition. Different prompt components compared to CODE4STRUCT are highlighted in yellow. We use event keywords from Li et al. (2021).

```
class Event:
    def __init__(self, name: str):
        self.name = name

class Movement(Event):
    def __init__(                   Parent Event Type
        self,
        agent: List[GPE | ORG | PER] = [],
        artifact: List[FAC | ORG | PER | VEH | WEA] = [],
        destination: List[FAC | GPE | LOC] = [],
        origin: List[FAC | GPE | LOC] = [],
        vehicle: List[VEH] = [],
    ):
        self.agent = agent
        self.artifact = artifact
        self.destination = destination
        self.origin = origin
        self.vehicle = vehicle
                                              Child event Transport
class Transport(Movement):                    inherit from parent Movement
    """self.agent transported self.artifact in self.vehicle
vehicle from self.origin place to self.destination place."""
    def __init__(
        self,
        agent: List[GPE | ORG | PER] = [],
        artifact: List[FAC | ORG | PER | VEH | WEA] = [],
        destination: List[FAC | GPE | LOC] = [],
        origin: List[FAC | GPE | LOC] = [],
        vehicle: List[VEH] = [],
    ):                                          Transport calls the __init__
        super().__init__(                       method of its parent Movement
            agent=agent,
            artifact=artifact,
            destination=destination,
            origin=origin,
            vehicle=vehicle,
        )
```

Figure A.4: Example of a hierarchical event definition. Different prompt components compared to CODE4STRUCT are highlighted in yellow.

```python
class Transaction(Event):
    def __init__(
        self,
        artifact: List[FAC | ORG | PER | VEH | WEA] = [],
        beneficiary: List[GPE | ORG | PER] = [],
        buyer: List[GPE | ORG | PER] = [],
        giver: List[GPE | ORG | PER] = [],
        place: List[FAC | GPE | LOC] = [],
        recipient: List[GPE | ORG | PER] = [],
        seller: List[GPE | ORG | PER] = [],
    ):
        self.artifact = artifact
        self.beneficiary = beneficiary
        self.buyer = buyer
        self.giver = giver
        self.place = place
        self.recipient = recipient
        self.seller = seller


class Transfer_Money(Transaction):
    """self.giver gave money to self.recipient for the benefit of
self.beneficiary in self.place place."""
    def __init__(
        self,
        beneficiary: List[GPE | ORG | PER] = [],
        giver: List[GPE | ORG | PER] = [],
        place: List[FAC | GPE | LOC] = [],
        recipient: List[GPE | ORG | PER] = [],
    ):
        super().__init__(
            beneficiary=beneficiary,
            giver=giver,
            place=place,
            recipient=recipient,
        )

class Transfer_Ownership(Transaction):
    """self.seller gave self.artifact to self.buyer for the benefit of
self.beneficiary at self.place place."""
    def __init__(
        self,
        artifact: List[FAC | ORG | PER | VEH | WEA] = [],
        beneficiary: List[GPE | ORG | PER] = [],
        buyer: List[GPE | ORG | PER] = [],
        place: List[FAC | GPE | LOC] = [],
        seller: List[GPE | ORG | PER] = [],
    ):
        super().__init__(
            artifact=artifact,
            beneficiary=beneficiary,
            buyer=buyer,
            place=place,
            seller=seller,
        )
```

Parent Event Type

Child event Transfer_Money inherit from parent Transaction

Transfer_Ownership is a sibling event type of Transfer_Money

Child event Transfer_Ownership inherit from parent Transaction

Figure A.5: Example of a hierarchical event definition with a sibling event type. Different prompt components compared to Figure A.4 are highlighted in yellow.

```
"""
Translate the following sentence into an instance of Transfer_Money. The
trigger word(s) of the event is marked with **trigger word**.
"If the budget goes through as is , why do n't Mr. Begala and Mr. Carville
just **donate** the extra tax money they do n't want ?"
"""
transfer_money_event = Transfer_Money(
    giver=[
        PER("Begala"),
        PER("Carville"),
    ],
)

"""
Translate the following sentence into an instance of Transfer_Ownership.
The trigger word(s) of the event is marked with **trigger word**.
"" The **acquisition** of Banco Zaragozano builds on our existing business
creating the sixth largest private sector banking group in Spain " by
assets , added Jacobo Gonzalez - Robatto , chief executive of Barclays
Spain ."
"""
transfer_ownership_event = Transfer_Ownership(
    artifact=[
        ORG("Banco Zaragozano"),
    ],
)
```

In-context example from sibling type **Transaction:Transfer_Money**

Make prediction for **Transaction:Transfer_Ownership**

Figure A.6: Example of a task prompt with a 1-shot example from sibling event type. Event definitions for the task prompt is shown in Figure A.5. Groundtruth prediction is colored green.

Parent Event Type	Child Event Type	# of Test Instances	# of Train Example
Business	Declare-Bankruptcy	2	39
	End-Org	5	24
	Merge-Org	0	13
	Start-Org	17	21
Conflict	Attack	90	1211
	Demonstrate	7	62
Contact	Meet	49	194
	Phone-Write	8	104
Justice	Acquit	1	4
	Appeal	6	30
	Arrest-Jail	6	72
	Charge-Indict	8	95
	Convict	6	61
	Execute	2	12
	Extradite	1	6
	Fine	6	22
	Pardon	0	2
	Release-Parole	1	44
	Sentence	11	83
	Sue	4	60
	Trial-Hearing	5	103
Life	Be-Born	3	44
	Die	17	516
	Divorce	9	20
	Injure	1	125
	Marry	10	71
Movement	Transport	47	561
Personnel	Elect	13	156
	End-Position	17	143
	Nominate	1	11
	Start-Position	11	87
Transaction	Transfer-Money	12	121
	Transfer-Ownership	27	85

Table A.2: The number of Train/Test event instances for 33 event types in ACE05-E.

Parent Event Type	Child Event Type (Train)	Child Event Type (Test)
Business	Declare-Bankruptcy	End-Org Merge-Org* Start-Org
Conflict	Attack	Demonstrate
Contact	Meet	Phone-Write
Justice	Trial-Hearing	Acquit Appeal Arrest-Jail Charge-Indict Convict Execute Extradite Fine Pardon* Release-Parole Sentence Sue
Life	Die	Be-Born Divorce Injure Marry
Personnel	Elect	End-Position Nominate Start-Position
Transaction	Transfer-Money	Transfer-Ownership

Table A.3: Train/Test split for each parent event type. * denotes child event types that do not have examples in the ACE05-E test set.