LLM-Based Java Concurrent Program to ArkTS Converter

Runlin Liu*
Beihang University
China
ler4065@gmail.com

Yuhang Lin*
Beihang University
China
yuhanglin35@gmail.com

Yunge Hu*
Beihang University
China
hygchn04@gmail.com

Zhe Zhang*
Beihang University
China
zhangzhe2023@buaa.edu.cn

Xiang Gao†
Beihang University
China
xiang_gao@buaa.edu.cn

ABSTRACT

HarmonyOS NEXT is a distributed operating system developed to support HarmonyOS native apps. To support the new and independent Harmony ecosystem, developers are required to migrate their applications from Android to HarmonyOS. However, HarmonyOS utilizes ArkTS, a superset of TypeScript, as the programming language for application development. Hence, migrating applications to HarmonyOS requires translating programs across different program languages, e.g., Java, which is known to be very challenging, especially for concurrency programs. Java utilizes shared memory to implement concurrency programs, while ArkTS relies on *message passing* (i.e., Actor model). This paper presents an LLM-based concurrent Java program to ArkTS converter.

Our converter utilizes large language models (LLMs) for efficient code translation, integrating ArkTS's SharedArrayBuffer API to create ThreadBridge, a library that replicates Java's shared memory model. Using LLM's Chain-of-Thought mechanism, the translation process is divided into specialized chains: the TS chain, concurrency chain, and synchronization chain, each handling TypeScript syntax, concurrency patterns, and synchronization logic with precision.

This study offers solutions to bridge concurrency model differences between Java and ArkTS, reducing manual code rewriting and speeding up adaptation for HarmonyOS NEXT. Experiments show the converter successfully compiles 66% of 53 test samples, with 69% accuracy for compiled results. Overall, the approach shows promise in converting concurrent Java programs to ArkTS, laying the foundation for future improvements.

KEYWORDS

Source code translations, HarmonyOS NEXT, ArkTS

*Authors listed in alpha-beta order.
†Corresponding Author

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10…$15.00
https://doi.org/10.1145/3691620.3695362

ACM Reference Format:
Runlin Liu, Yuhang Lin, Yunge Hu, Zhe Zhang, and Xiang Gao. 2024. LLM-Based Java Concurrent Program to ArkTS Converter. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3691620.3695362

1 INTRODUCTION

HarmonyOS NEXT introduces a new programming environment utilizing ArkTS, a superset of TypeScript, to develop native applications [5]. Migrating applications from Android to HarmonyOS is challenging due to differences in concurrency models. Java uses a shared memory model where threads communicate through shared variables, while ArkTS adopts an actor-based model, where actors communicate via message passing without shared memory. [4]

Transitioning from Java's shared memory model to ArkTS's actor model is challenging due to the absence of direct equivalents for thread interactions and synchronization. Manual translation is time-consuming and error-prone, particularly for large codebases with complex concurrency patterns.

Existing code translation tools primarily focus on syntactic transformations and fall short when it comes to translating complex concurrency constructs. Traditional Abstract Syntax Tree (AST) conversion techniques struggle with handling complex constructs such as inner classes and concurrency. Tools like JSweet [1], which translate Java to TypeScript, do not support concurrency programs, require additional software packages, and only support traditional TypeScript, not ArkTS. These tools are not suited for new languages like ArkTS and fail to address the deep semantic transformations needed for concurrency model conversion. Recent studies show LLMs' potential for automated code translation, with Yang et al. [10] making significant strides. However, translating Java to ArkTS with LLMs is challenging due to the lack of high-quality ArkTS training data, leading to incomplete grasp of its syntax and semantics.

To address the complexities of migrating Java applications to HarmonyOS's ArkTS, we introduce the Java2ArkTS source code converter, which ingeniously recreates Java's shared memory and synchronization mechanisms within the ArkTS framework. To implement concurrency in ArkTS with a Java-like structure, our tool provides a custom concurrency library. This library enables shared memory-like interactions, which ArkTS lacks natively, allowing developers to write concurrent programs using familiar Java constructs and paradigms. The core of this library is the SharedArrayBuffer provided by native ArkTS(TS), which allows true shared

memory across multiple threads. [3] [8] Unlike traditional ArkTS arrays, SharedArrayBuffer enables our custom concurrency library to implement shared memory capabilities, crucial for supporting complex synchronization and communication patterns in multi-threaded programming.

Central to our methodology is the utilization of LangChain's cognitive chain technology, which orchestrates the translation process into a coherent sequence of well-defined steps. [2] This cognitive chain, underpinned by the robust understanding of TypeScript by Large Language Models (LLMs), meticulously navigates the intricacies of semantic transformation from Java to ArkTS. By leveraging the close relationship between TypeScript and ArkTS and the advanced prompt engineering techniques [6], the Java2ArkTS converter ensures a precise and efficient translation.

The contributions of this paper are summarized as follows:

(1) Simulating Shared Memory in ArkTS: To translate Java's shared memory concurrency model, we use ArkTS's SharedArrayBuffer and a custom library that replicates Java's communication and synchronization interfaces, preserving the original concurrent behavior in ArkTS.

(2) Leveraging TypeScript and Advanced Prompt Engineering: Utilizing the extensive understanding of TypeScript by LLMs, we employ advanced prompt engineering techniques along with chain-of-thought methodology. [9] This approach breaks down the translation process into manageable steps, ensuring accurate and efficient conversion of Java's complex concurrency constructs into ArkTS.

The source code, datasets, evaluation materials, and demo video can be accessed at: https://github.com/Java2ArkTS/Java2ArkTS

2 PROJECT USAGE AND ITS ABILITY

The user interaction interface of the source code converter is accessible through a web browser.

Users can upload Java source files, track the translation process, and reviewresults directly in a user-friendly browser interface. The converter handlescomplex Java structures, such as inner classes and concurrency, ensuring semantic and structural accuracy, where is better thantools like JSweet.

3 TECHNICAL DETAILS

The workflow of our tool is shown in Figure 1. Firstly, in order to simplify the code conversion issue, we independently design a ThreadBridge library using the API provided by ArkTS. The ThreadBridge library mimics multi-threaded operations in the Java language using ArkTS language and provides corresponding APIs. Secondly, as the LLM cannot solve such a complex problem in one step, we decompose the transformation problem into a set of sub-problems. The mechanism guides the LLM to use the ThreadBridge library for conversion. If the transformed code fails to compile, the error is sent back for correction, with a limit on iterations to prevent loops.

3.1 Memory Sharing Under the Actor Model

The ArkTS concurrency model, unlike Java, follows the Actor model, which lacks support for memory sharing. To simplify the translation process, we propose to implement a library using SharedArrayBuffer to mimic memory sharing. SharedArrayBuffer is an API used

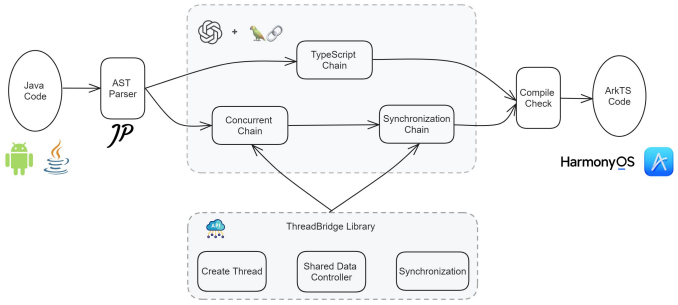

Figure 1: Overall architecture of Java2ArkTS. The Java code is preprocessed by Java Parser. TypeScript-like code is converted via the TypeScript chain. Concurrent code is translated using our ThreadBridge shared library, and processed through the concurrency and synchronization chains. Finally, we merge the conversion results and check for compilation errors, producing the converted ArkTS concurrent code.

in TypeSrcipt to create shared memory. It allows multiple threads to share the same memory area, enabling parallel computing and efficient data sharing. However, assigning values to SharedArrayBuffer is intricate since SharedArrayBuffer, being a generic buffer, does not store data with their corresponding type. However, ArkTS is a statically typed programming language, which requires developers to explicitly declare the data types of variables when they are defined. It is almost impossible for LLM to implement translation between generic bufferes with certain types. Hence, we design the ThreadBridge library to simplify the work of LLM.

Specifically, in ThreadBridge, we implement functions for each basic type to translate it into a sharedArrayBuffer object, then combine these functions into getShared to convert all types to sharedArrayBuffer objects. For instance, Listing 1 shows an example that converts data of type number into a sharedArrayBuffer Object. It first creates a sharedArrayBuffer, turns it into typed arrays, assigns values to typed arrays, and finally returns this typed array and type marker. Afterward, we design a reader called `getXValues` to read the values of the sharedArrayBuffer object. This function first detects the type of the incoming data. If it is a basic type, it calls the corresponding type's function to read its value. In contrast, if it is an object, the `getXValues` will traverse its members and recursively call `getXValues`. We also design `setXValues` to assign values to sharedArrayBuffer objects, which is similar to `getXValues`.

After implementing memory sharing, we need to solve the problem of synchronization. We design function `synStart` to get locks, and `synEnd` to release locks, they are based on Atomics. Atomics is a class in ArkTS which provides a set of static methods for performing atomic operations on SharedArrayBuffer objects. We simulate Java's synchronized by combining Atomics and SharedArrayBuffer.

In ArkTS, we implement the Thread class and Runnable interface similarly to Java. The Thread class can accept objects that implement the Runnable interface, and invoking the `start` function initiates the thread. We utilize the taskpool API in ArkTS to manage thread creation.

However, taskpool serializes parameters at runtime and loses member methods during this process. Therefore, we can't simply

```ts
function generateNumberShared(n ?: number) {
    const buffer = new SharedArrayBuffer(8);
    const sharedArray = new Float64Array(buffer);
    sharedArray[0] = 0;
    if (n != null) {
        sharedArray[0] = n;
    }
    return {
        "sharedValue": sharedArray,
        "sharedType": "number",
    };
}
```

Listing 1: An example of `generateNumberShared`

pass the runnable objects as a parameter to taskpool. Instead, we store all runnable objects as a static member of Thread class, and provide the index of runnable objects that need to be run to taskpool.

3.2 Program Translation via Chain-of-Thought

The core of the JAVA2ARKTS lies in using LangChain tool to create a chain-of-thought. The prompt method is essential as it doesn't require extensive training data. However, large language models struggle with new languages like ArkTS. Hence, JAVA2ARKTS first decomposes the task of code conversion into multiple subtasks, forming thought chains. These chains can be divided into three parts: **TS chain** - converts Java code into TypeScript- specific syntax in ArkTS code; **Concurrent Chain** - converts Java concurrency semantics using the API in the ThreadBridge library for multi-threaded shared memory; **Sync Chain** - converts synchronization code using the API in the ThreadBridge library. In order to make the conversion results more accurate, we separate the Java source code by class and guide the LLM to convert them one by one.

Using the same set of prompt words to convert Java code is inefficient and may result in errors. For example, if we prompt the LLM to change the usage of synchronized in functions that do not exist, the large model is likely to add extra usage, but this is incorrect and may not be easily detected in future checks. To prevent incorrect modification by LLM and ensure the accurate targeting of code segments by LLM, we employ the Abstract Syntax Tree (AST) analysis, which is a mature and easy-to-use method for analyzing the structure of Java code [7]. We use AST to analyze each class in Java code and select the appropriate thought chain for it. Figure 2 shows the general process of selecting a thought chain.

3.2.1 TS Chain. TS chain conversion does not involve concurrent code. ArkTS is a superset of TypeScript and LLMs are very familiar with TypeScript, which allows for accurate conversion of Java code that does not involve concurrency into TypeScript code, allowing ArkTS to adapt to these TypeScript codes. In more complex cases, due to our class-by-class conversion approach, the LLM lacks contextual information at this stage, which may lead to errors when independently writing other classes used in the code. Therefore,

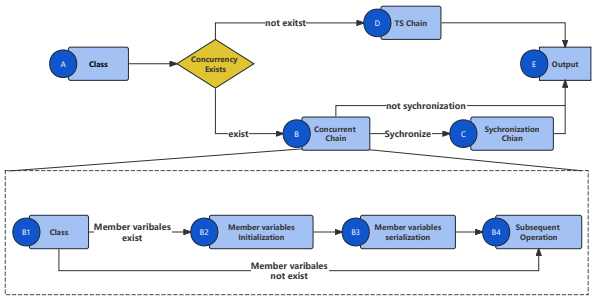

Figure 2: Analyze code features through AST and select appropriate thought chains based on the analysis results.

we prompt the LLM to summarize all Java code into contextual information and provide it in every subsequent prompt. This method is also used in other chains.

3.2.2 Concurrent Chain. The Concurrent Chain is the focus of this project, which achieves memory sharing by enabling LLM to understand and use the ThreadBridge library one by one. First, we prompt the LLM to understand and use getValues and setValues. To ensure the LLM uses these functions more accurately, we assign specific roles to the LLM at the beginning of the prompt, which has proven effective in prompt engineering. When explaining a function, the prompt not only introduces its usage and roles but also provides a few examples to help the LLM understand. Part of the prompt is shown in Figure 3.

Basic Prompt for GPT

```
If you are a programmer who is very good at typescript.
Please change the user input classes with the following functions.
The existing function getValues(obj: any) is passed to a member variable of type any and returns its corresponding value.
```

Sample Prompt for GPT

```
input:   console.log(this.s);
output:  console.log(getValues(this.s));
input:   a = this.test.getNum();
output:  a = getValues(this.test.getNum());
```

Figure 3: For some prompts in the concurrent chain, first assign roles to LLM, then introduce the usage and role of getValues to LLM, and finally provide a few examples to enhance understanding.

Then, we prompt the LLM to understand and use getShared. However, the getShared function requires an instance of the object to be passed in, so all member variables need to be initialized first. During the initialization process, the LLM needs to know the constructor methods of member variables to correctly create instances without causing compilation errors. To address this, we design additional sub-chains to prompt LLM to obtain the required constructor information. Unlike the context summary mentioned earlier, this prompt requires more precise and targeted output. Even with significant adjustments made to prompt words, LLM may still misuse these functions. Therefore, we increase the robustness of the ThreadBridge library to tolerate most misuse.

3.2.3 Syn Chain. The Sync Chain utilizes the API provided by the ThreadBridge library to complete the conversion of the synchronization part of concurrent programs. Initially, with the capabilities of LLM, it replaces the synchronized keyword with synStart and synEnd, ensuring these functions encase the code blocks that were originally delineated by synchronized. To manage locking positions accurately, each class requires a unique identifier for every synchronized block. Consequently, we instruct the LLM to enumerate the synchronized blocks and generate a corresponding index. This index is then passed to the synStart and synEnd functions, thereby accomplishing the conversion of the synchronization code segment.

4 EXPERIMENTAL RESULTS

4.1 Experimental Setting

To validate our converter, we curated 53 Java concurrency code samples based on classical synchronization patterns like producer-consumer and reader-writer, ranging from easy to hard levels. We used JRE and DevEco Studio (HarmonyOS IDE with ArkTS) to execute and verify the code before and after conversion, using stable API version 9 for testing to ensure consistent results.

4.2 Evaluation Metrics

Using the dataset above, we transformed Java concurrency code into ArkTS and verified the correctness of the concurrency behavior in the generated ArkTS programs to confirm the fidelity of the conversion process. We executed the Java and ArkTS code before and after conversion, manually assessed their logical consistency, and evaluated the results based on successful compilation and correct execution. Also, we analyzed the causes of compilation failures or runtime errors.

4.3 Results

Among the 53 test samples, 66% of the results pass compilation, and the accuracy rate is 69% in the successfully compiled results. The most common error in compilation is the incorrect use of the ThreadBridge library, since the LLM struggle with additional requirements, even though the usage of the function has been described in detail, the LLM may still use it in the wrong place or pass invalid parameters to it. Adding extra things refers to the fact that during the transformation process, the LLM does not recognize the functions or classes provided in the context and writes new functions and classes on its own, which conflicts with the context.

Table 1: Success Rate of compiling and running

	Successful Quantity	Failed Quantity	Success Rate
Compiling	35	18	66.0%
Running	24	29	45.3%

Our runtime error analysis revealed that nearly two-thirds of errors were due to the LLM's incorrect use of the ThreadBridge library, particularly in complex scenarios. However, for programs with moderate complexity and standardized coding, our approach effectively supports accurate conversions, demonstrating reliability in structured environments.

Table 2: Causes of Compiling Failures and Incorrect Running

	Incorrect ThreadBridge Using	Adding Extra Things	Incorrect Initialization	Others
Compiling	47.4%	10.5%	10.5%	31.6%
Running	63.6%	0	0	36.4%

5 LIMITATION

Currently, our converter struggles with code that relies on various imported libraries, making it less effective for real-world applications using different libraries in Java and ArkTS. The low conversion success rate could be improved by using techniques like Automatic Program Repair (APR) to fix translation errors or optimizing LLMs to reduce hallucinations and enhance the overall approach.

6 CONCLUSION

This paper introduces a LangChain-based converter for translating concurrent Java programs to ArkTS, the language of HarmonyOS NEXT, using large language models (LLMs) to tackle challenges in different concurrency models. Experiments show a 66% compilation success rate and 69% accuracy among compiled results from 53 test samples, with common issues including ThreadBridge misuse and unnecessary elements added by the LLM. Despite these challenges, the converter shows promise in automating Java-to-ArkTS translation, reducing manual work and speeding up HarmonyOS NEXT deployment. Future efforts will aim to enhance accuracy and support for various concurrency patterns.

ACKNOWLEDGMENTS

This work was supported by National Natural Science Foundation of China under Grant Nos (62202026, 62141209).

REFERENCES

[1] 2023. Jwseet. https://github.com/cincheo/jsweet Accessed: 2024.
[2] 2024. LangChain. https://github.com/langchain-ai
[3] Andreas Costi, Brian Johannesmeyer, Erik Bosman, Cristiano Giuffrida, and Herbert Bos. 2022. On the effectiveness of same-domain memory deduplication. In Proceedings of the 15th European Workshop on Systems Security. 29–35.
[4] Nathaniel Dempkowski. [n. d.]. Message Passing and the Actor Model. ([n. d.]).
[5] Li Li, Xiang Gao, Hailong Sun, Chunming Hu, Xiaoyu Sun, Haoyu Wang, Haipeng Cai, Ting Su, Xiapu Luo, Tegawendé F Bissyandé, et al. 2023. Software Engineering for OpenHarmony: A Research Roadmap. arXiv preprint arXiv:2311.01311 (2023).
[6] Ggaliwango Marvin, Nakayiza Hellen, Daudi Jjingo, and Joyce Nakatumba-Nabende. 2023. Prompt Engineering in Large Language Models. In International Conference on Data Intelligence and Cognitive Informatics. Springer, 387–402.
[7] Javaparser Project. [n. d.]. javaparser. https://github.com/javaparser/javaparser
[8] Conrad Watt, Christopher Pulte, Anton Podkopaev, Guillaume Barbier, Stephen Dolan, Shaked Flur, Jean Pichon-Pharabod, and Shu-yu Guo. 2020. Repairing and mechanising the JavaScript relaxed memory model. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. 346–361.
[9] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. Advances in neural information processing systems 35 (2022), 24824–24837.
[10] Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. 2024. Exploring and unleashing the power of large language models in automated code translation. arXiv preprint arXiv:2404.14646 (2024).