RESEARCH STATEMENT

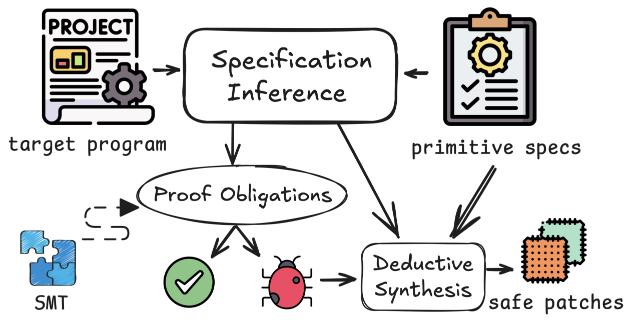
Yahui Song
Research Fellow at National University of Singapore (NUS), Singapore
https://www.comp.nus.edu.sg/~yahuis/

My research focuses on Programming Languages and Formal Methods, emphasizing Specification, Compositional Verification, and Logic – particularly temporal and separation logic. In general, "compositional verification" refers to the capability of proving a program's correctness or incorrectness in a modular manner, which decomposes the reasoning about a whole, complex program into an analysis of its simpler, reusable components.

I am especially interested in leveraging formal reasoning techniques for establishing program correctness, bug detection, and automated program repair. These methods complement traditional approaches like testing, which often fail to prove the absence of bugs in high-critical systems. For the future of program analysis and software reasoning, I believe the focus should be on the following key areas: (I) Advanced Logic for Specification: Developing sophisticated logical frameworks to enhance specification capabilities; (II) Practical Inference Mechanisms: Creating effective methods for generating modular specifications; (III) Automated Formal Verification Frameworks: Streamlining the process of proving software correctness through automation; and (IV) Scalability: Ensuring that systems support automated, incremental reasoning and promote the reuse of effort.

Past Research and Ongoing Work

Modular & Expressive Temporal Verification. The predominant technique in temporal verification, automata-based model checking, has several limitations: it requires a manual modelling phase, is constrained by bounded analysis due to the lack of symbolic reasoning, and is restricted in expressiveness by finite-state automata. To address these challenges and combine the advantages of both modularity and automation, I explore a more precise, extensive, and efficient solution for temporal verification. More specifically, it deploys Hoare-style forward verifiers as the front end and term rewriting systems (TRSs) as the back end. Forward verifiers compute the actual temporal behaviours from the source code based on the formally defined execution semantics of the target languages. TRSs are decision procedures inspired by Antimirov and Mosses's algorithm but solving the language inclusions between the actual behaviours and specifications, both written in expressive *symbolic linear temporal properties*. In this approach, the temporal constraints can be dependent on program variables, allowing for more expressive and precise verification. My PhD thesis demonstrates the applications of this framework across various domains, including event-based reactive systems [1], synchronous languages like Esterel [2], user-defined algebraic effects and handlers [3], real-time systems [4], and preemptive asynchronous programs [5].



Program Repair guided by Linear Temporal Properties. As a postdoctoral fellow, I expanded my research into bug detection and automated program repair. To enable a more flexible specification style on top of the classic pre/post-conditions, our recent work [6] allows users to write a "future condition" to modularly express the expected behaviours after the function call, which provides the guideline for automatically detecting and repairing temporal bugs. Our tool, ProveNFix, is configured with 17 pre-defined specifications for primitive APIs and detects 515 bugs from over 1 million lines of code spanning ten real-world C projects. The benefit of our approach is that a small set of properties can be specified once and used to analyze a large number of programs. The novel analysis contributes to an efficient repair strategy that generates patches symbolically. This work has been recognized with a **Distinguished Paper Award at FSE 2024** for its advancements in automated *specification* *inference* and its broad applicability across diverse types of bugs. The powerful specification inference techniques help understand complex stateful implementations, for which often no formal specification may be available in real life - partially addressing a long-standing issue in software engineering.

Staged Specification Logic. I am broadly interested in language features such as higher-order imperative programming, which promotes reusable code and design patterns, and algebraic effects, which provide a monadic framework for handling non-determinism and probabilistic programming. To verify such essential language features, one inherent limitation of existing specification mechanisms is their reliance on only two stages: an initial stage to denote the precondition at the start of the method and a final stage to capture the postcondition.

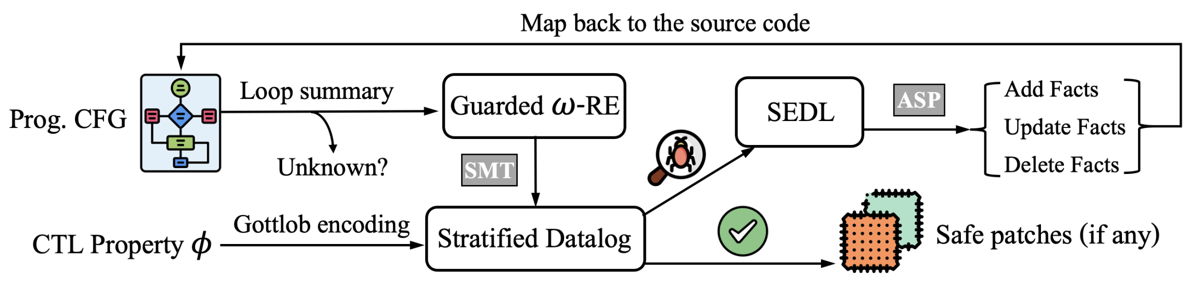
Bi-Abduction

$$\frac{\texttt{?A * P} \vdash \texttt{Q * ?F}}{\texttt{ens P; req Q} \longrightarrow \texttt{req A; ens F}}$$

Such two-stage specifications force abstract properties to be imposed on unknown function parameters, leading to less precise specifications for higher-order methods. To overcome this limitation, we introduce a novel extension to Hoare logic that supports multiple staged specifications with a separation logic instantiation. Multiple stages allow the behaviour of unknown function-type parameters/unhandled effects to be captured abstractly as uninterpreted relations. Staged logic is supported by a set of *bi-abduction*-based normalization, which allows a mostly automated specifications inference, requiring only auxiliary lemmas to be provided. Our "staged specification" has proven effective in verifying higher-order imperative programs [7], heap-manipulating algebraic effects and handlers [8], and shift/reset control operators [9].

Incorrectness Proofs for Object-Oriented Programs. Inheritance and method overriding are crucial concepts in object-oriented (OO) programming languages. These concepts support a hierarchy of classes that reuse common data and methods. However, techniques to reason about the incorrectness of OO programs are yet to be investigated. We first present a mechanism [10] that 1) specifies the normal and abnormal executions of OO programs by using ok and err specifications, respectively; 2) verifies these specifications by a novel under-approximation proof system based on incorrectness logic (IL). Subsequently, we present an IL specification inference system [11] that automatically generates specifications and facilitates a push-button bug detection tool. At its core, we encode type information into bi-abductive reasoning and propagate type constraints throughout the analysis. The direct benefit is that we can efficiently identify bugs caused by improper usage of the casting operator, which cannot be handled by the existing techniques. Meanwhile, our system can reduce false positives while finding more true bugs because of not losing OO-type information. Experimental results show that our proposal finds 400% more class-cast-exceptions compared with *Error Prone* (developed by Google) and improves the precision of finding null-pointer-exceptions by 24.4% compared with *Pulse* (developed by Meta).

Program Repair guided by CTL Properties. My interests also encompass *Computational Tree Logic* (CTL), which is founded on a branching concept of time, i.e., at each moment, there may be several different possible futures. Many CTL properties, such as reachability, termination, invariants, and responsiveness, are commonly specified in infinite-state programs. We suggest a mechanism for the automated repair of real-world programs guided by CTL properties. Our repair framework [12] is based on Datalog, a widely used logical inference language for program analysis, which readily supports nested CTL via stratified negation. Specifically, we encode the program and CTL properties into Datalog facts and rules and perform the repair by modifying the facts to pass the analysis rules. The repair is discharged by an extended *Symbolic Execution of Datalog* (SEDL). This work



focuses on proving both safety and liveness properties where loops are handled via a novel summarization, represented by a *guarded ω-regular language*. Unlike existing loop summaries, which do not explicitly capture non-terminating behaviours, we capture both terminating and non-terminating behaviours in a (guarded) disjunctive form. Given the undecidable nature of termination analysis, it outputs "Unknown" when there exists a path for which we cannot conclusively prove either termination or non-termination. This work advances existing repair techniques to encompass analyses defined by both least-fixpoint and greatest-fixpoint semantics.

Plans for Future Research

I envision several key directions for future work that will enhance the capabilities of program analysis and automated reasoning. These directions align with the overarching goal of improving software correctness and reliability in complex systems, as well as advancing logic-augmented generation for large language models (LLMs).

Large Scala Temporal Verification via Precise Loop Summaries. Termination is a sub-problem of temporal properties because it directly influences the ability to ensure that a program eventually reaches a desired state or outcome. I propose several concrete directions for achieving scalable temporal verification via loop summaries.

(1) **Large Scala (Non-) Termination Analysis.** Many real-world programs still suffer from vast consequences caused by non-termination bugs. While various termination-checking tools have proven effective on well-established benchmarks, recent studies [13] indicate that these tools must be more effective with real-world projects and benchmarks. Therefore, existing termination analysis tools must be enhanced to improve their scalability and applicability to real-world projects. Furthermore, errors in general programming features, such as bit-level arithmetic, heap usage and recursive functions, etc., should be paid more attention to in future termination analysis research. A combination of my existing research on compositional verification and effective loop summarization — for both terminating and non-terminating behaviours — can help overcome the limitations of existing tools and contribute to a more robust termination analysis.

(2) **More Precise Safety Checking.** My previous work introduced ProveNFix, the first compositional temporal analysis tool designed for large-scale programs [6]. It detects general safety property violations, including null pointer dereferences, resource leaks, and memory leaks. However, similar to many analysis tools that rely on incorrectness logic [14, 10], ProveNFix handles loops via unrolling. Such a design choice results in many false negatives, i.e., undetected bugs, in both theory and practice, as it under-approximates loop behaviour by considering only a finite number of execution instances. I believe these false negatives can be significantly reduced by employing more accurate loop summaries derived from the methods in (1).

(3) **Large Scala Liveness Checking.** Verifying liveness properties in protocol implementations is essential for ensuring progress, avoiding starvation, verifying interactions, etc. Existing works for capturing liveness bugs are based on grey-box fuzzing [15]. While testing-based approaches like fuzzing are effective, they often struggle to prove the absence of bugs. Therefore, I plan to pursue the first modular liveness-checking tool for protocol implementations. The key insight is that most liveness properties can be reduced to a set of safety properties with the help of ranking functions [16]. More specifically, given any global liveness property and a large codebase that contains many function definitions, a specification inference process projects the global (liveness) requirement to individual functions, and such a projection creates modular (safety) specifications for each function. These generated specifications can be further used for bug detection/repair.

Advanced Logic for Specification and Practical Inference Mechanisms. Staged specification [7, 8] involves decomposing a program's specification into multiple stages, each focusing on different aspects or levels of abstraction. This approach allows for more granular and modular reasoning about different components or phases of the program's execution. The capabilities of staged specifications have the potential to benefit more language features such as asynchronous/concurrent programs, other implementations for effects handlers, etc.

(1) **Staged Specification for Concurrency.** Asynchronous programs encompass concurrent operations, including asynchronous I/O, parallel tasks, and event-driven programming. Managing the interplay between these operations can be challenging to specify and verify due to their inherent complexity and non-deterministic nature. The staged specification offers a solution by dividing the specification into distinct stages, each addressing specific segments of the program's behaviour. This approach is especially beneficial for verifying preemptive asynchronous programs with complex execution semantics, providing a structured method to navigate their intricate interactions and ensure correctness.

(2) **Staged Specification for Low-level Code.** WebAssembly (Wasm) is a low-level, portable code format that delivers near-native performance and serves as a compilation target for various source languages. Previously, Wasm lacked direct support for non-local control flow features like async/await, generators, and lightweight threads. However, recent advancements introduced WasmFX [17], which provides a universal target for these features through effect handlers, enabling compilers to translate them directly into Wasm. Notably, the use of staged specifications aids in verifying low-level abstractions in WebAssembly, necessitating a deeper understanding of its execution model, including stack operations and memory management.

(3) **Lemma Synthesis for Staged Specification.** As previously noted, staged specifications benefit from automated inference, requiring only minimal specifications such as auxiliary lemmas. However, crafting these lemmas can be challenging due to their complexity and the nuanced understanding needed to ensure correctness. To address this issue, employing a search-based or data-driven approach that utilizes input-output examples could significantly enhance the automation of the verification process. These methods streamline lemma generation by leveraging existing data and patterns, thereby reducing the need for manual effort.

Expanding the Expressive Power of Future Conditions. The existing application for future conditions [6] has demonstrated effectiveness in specification inference and bug detection, guided by temporal logic properties such as "finally, the allocated memory will be freed" and "globally, there is no null pointer dereference." However, its reliance on pure arithmetic and temporal constraints represented as extended regular expressions limits its ability to analyze complex aliasing and reassignment operations. While [6] attempts to mitigate false positives caused by these limitations, it does so at the expense of true positives. To address this issue, I plan to enhance the expressiveness of future conditions by incorporating separation logic. This approach will enable better tracking of mutations to local and global variables as well as the aliasing relationships among these variables.

Detecting and Reducing Hallucinations in LLMs Through Logical Reasoning. It is known that LLMs struggle with generating hallucinations, i.e., coherent yet factually inaccurate outputs. A major concern is fact-conflicting hallucination (FCH), where LLMs produce content contradicting ground truth facts. The existing work [18] addresses this issue by creating a testing framework that builds a factual knowledge base from sources like Wikipedia. It uses logical reasoning rules to transform and augment this knowledge into a large set of test cases with ground truth answers. However, its capability to evaluate the complex logical reasoning of LLMs is limited, as its rules only cover simple relations like negation and symmetry, etc. Meanwhile, LLMs still struggle with more complex logical reasoning, such as temporal reasoning, due to the lack of understanding of time and the semantics of temporal operators. To detect such temporal hallucinations, I propose to automatically generate test cases that effectively ensure a more robust evaluation of the LLMs' ability to handle reasoning tasks and identify factual inconsistencies. By applying reasoning rules derived from temporal logic, one can mutate and expand the initial seed data from the knowledge base, enhancing the diversity and complexity of the test scenarios.

To further mitigate temporal hallucinations, I plan to integrate formal logic into large language models (LLMs) through "Logic-Augmented Generation," which aims to improve their temporal reasoning capabilities. Specifically, if LLMs can effectively decompose the components of a temporal query — such as temporal operators and atomic propositions — we could instrument the temporal reasoning process with formal methods like Prolog.

This approach would enhance the correctness and reliability of responses, with any uncertainty arising only from the decomposition phase, while ensuring that the temporal reasoning remains sound and complete.

References

[1] **Yahui Song** and Wei-Ngan Chin. Automated temporal verification of integrated dependent effects. In *Formal Methods and Software Engineering: 22nd International Conference on Formal Engineering Methods, ICFEM 2020*, pages 73–90. Springer, 2020.

[2] **Yahui Song** and Wei-Ngan Chin. A synchronous effects logic for temporal verification of pure esterel. In *Verification, Model Checking, and Abstract Interpretation: 22nd International Conference, VMCAI 2021*, pages 417–440. Springer, 2021.

[3] **Yahui Song**, Darius Foo, and Wei-Ngan Chin. Automated temporal verification for algebraic effects. In *Programming Languages and Systems - 20th Asian Symposium, APLAS 2022, Proceedings*, volume 13658 of *Lecture Notes in Computer Science*, pages 88–109. Springer, 2022.

[4] **Yahui Song** and Wei-Ngan Chin. Automated verification for real-time systems - via implicit clocks and an extended antimirov algorithm. In *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Proceedings, Part I*, volume 13993 of *Lecture Notes in Computer Science*, pages 569–587. Springer, 2023.

[5] **Yahui Song** and Wei-Ngan Chin. Automated temporal verification for preemptive asynchronous programs (ongoing work).

[6] **Yahui Song**, Xiang Gao, Wenhua Li, Wei-Ngan Chin, and Abhik Roychoudhury. ProveNFix: Temporal property-guided program repair. *Proc. ACM Softw. Eng.*, 1(FSE):226–248, 2024. **Distinguished Paper Award**.

[7] Darius Foo, **Yahui Song**, and Wei-Ngan Chin. Staged specifications for automated verification of higher-order imperative programs. *FM 2024*.

[8] **Yahui Song**, Darius Foo, and Wei-Ngan Chin. Specification and verification for unrestricted algebraic effects and handling. *Proc. ACM Program. Lang.*, 8(ICFP), aug 2024.

[9] Darius Foo, **Yahui Song**, and Wei-Ngan Chin. A typed and cps-enabled hoare logic for shift/reset (ongoing work).

[10] Wenhua Li, Quang Loc Le, **Yahui Song**, and Wei-Ngan Chin. Incorrectness proofs for object-oriented programs via subclass reflection. In *Programming Languages and Systems - 21st Asian Symposium, APLAS 2023*, volume 14405 of *Lecture Notes in Computer Science*, pages 269–289. Springer, 2023.

[11] Wenhua Li, Quang Loc Le, **Yahui Song**, and Wei-Ngan Chin. Inferring incorrectness specifications for object-oriented programs (under submission).

[12] Yu Liu*, **Yahui Song***, Martin Mirchev, Sergey Mechtaev, and Abhik Roychoudhury. Computation tree logic guided program repair with precise loop summaries (under submission).

[13] Xiuhan Shi, Xiaofei Xie, Yi Li, Yao Zhang, Sen Chen, and Xiaohong Li. Large-scale analysis of non-termination bugs in real-world OSS projects. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022*, pages 256–268. ACM, 2022.

[14] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. Finding real bugs in big programs with incorrectness logic. *Proc. ACM Program. Lang.*, 6(OOPSLA1):1–27, 2022.

[15] Ruijie Meng, Zhen Dong, Jialin Li, Ivan Beschastnikh, and Abhik Roychoudhury. Linear-time temporal logic guided greybox fuzzing. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022*, pages 1343–1355. ACM, 2022.

[16] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. Mostly automated verification of liveness properties for distributed protocols with ranking functions. *Proc. ACM Program. Lang.*, 8(POPL):1028–1059, 2024.

[17] Luna Phipps-Costin, Andreas Rossberg, Arjun Guha, Daan Leijen, Daniel Hillerström, K. C. Sivaramakrishnan, Matija Pretnar, and Sam Lindley. Continuing webassembly with effect handlers. *Proc. ACM Program. Lang.*, 7(OOPSLA2):460–485, 2023.

[18] Ningke Li, Yuekang Li, Yi Liu, Ling Shi, Kailong Wang, and Haoyu Wang. Drowzee: Metamorphic testing for fact-conflicting hallucination detection in large language models. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):1843–1872, 2024.