Construction of a Memory Safety Model for Concurrent Programs Based on Formal Verification

Hao Su*

Hainan Vocational University of Science and Technology, Haikou, 571126, China
**Corresponding author: suhao2580@126.com*

Abstract: *As multi-core processors become the mainstream in computing, the memory safety issues of concurrent programs have grown increasingly severe due to their nondeterministic interactions. Formal verification, which provides rigorous guarantees for program correctness through mathematical methods, serves as a key approach to addressing this challenge. This paper aims to systematically construct a memory safety model for concurrent programs based on formal verification. The research first clarifies the formal definitions of spatial and temporal safety in concurrent environments and precisely characterizes shared memory interactions based on interleaving semantics and operational semantics. Subsequently, a modeling approach oriented toward concurrent memory safety is proposed, including abstract state machine modeling, logical representation of execution traces, and an extended separation logic resource specification framework. Finally, the study investigates automatic construction and refinement strategies for model invariants, explores verification algorithms such as symbolic execution and abstract interpretation, and designs a modular, extensible verification framework. This research provides a comprehensive technical pathway from theory to methodology for the systematic verification of concurrent memory safety.*

Keywords: *Formal Verification; Concurrent Programs; Memory Safety; Separation Logic; Model Construction; Symbolic Execution; Abstract Interpretation*

Introduction

Against the backdrop of the ever-increasing complexity of software systems, concurrent programming has emerged as a crucial technology for achieving high performance and responsiveness. However, concurrent access to shared memory introduces issues such as data races, atomicity violations, and order violations. These interaction defects can directly lead to subtle yet critical memory safety violations, including buffer overflows and dangling pointer dereferences, which pose significant threats to system reliability and security. Due to the highly non-deterministic and scenario-dependent nature of concurrency errors, traditional methods like dynamic testing and code review possess inherent limitations and struggle to exhaust all possible thread interleavings. Consequently, advancing formal verification techniques capable of providing rigorous mathematical proofs of program correctness holds urgent research significance and necessity for constructing high-assurance concurrent systems. The research presented in this paper precisely focuses on how to systematically construct a formal verification model tailored for the memory safety properties of concurrent programs. This model must not only provide a precise description of complex memory safety properties—encompassing both spatial and temporal dimensions—and their variations within the concurrent context but also establish a complete methodological framework. This framework spans from program semantic characterization and formal specification to automated verification algorithms, aiming to address core challenges such as state space explosion and verification complexity. Ultimately, it seeks to lay a solid theoretical foundation and provide a feasible technical solution for achieving reliable guarantees of memory safety in concurrent programs.

1. Formal Theoretical Foundations for Memory Safety in Concurrent Programs

1.1 Formal Definition and Classification of Memory Safety Properties

The core of memory safety properties lies in ensuring that every memory access performed by a program occurs within its permitted range and that the type of access conforms to the semantics defined

for that memory region. Formal methods employ precise mathematical language to characterize these properties, typically decomposing memory safety into two orthogonal dimensions: spatial safety and temporal safety. Spatial safety concerns the validity of pointer references, ensuring that pointer dereference operations do not exceed bounds or access unallocated memory units. Temporal safety pertains to the lifecycle of memory objects, prohibiting access to memory regions that have already been freed, thereby eliminating dangling pointer dereferences. These two categories of properties can be formally expressed using rules from predicate logic or type systems, such as employing assertions in separation logic to describe exclusive ownership of memory regions, or encoding memory addresses and lifecycle markers within types.

A systematic classification of memory safety properties serves as the prerequisite for constructing targeted verification models. The fundamental classification can be conducted based on the severity of violation consequences and the timing of detection. For instance, memory leaks, buffer overflows, and double frees can be categorized as observable runtime errors. A deeper level of classification distinguishes properties based on their degree of coupling with concurrent interactions, separating so-called "concurrent memory safety" properties—those that hold in single-threaded environments but may be violated under interleaved concurrent execution. The formalization of such properties requires additional consideration of consistency constraints governing the transfer and evolution of memory states across multiple execution threads, thereby laying the groundwork for subsequent concurrency-specific modeling[1].

1.2 Semantic Characterization of Concurrent Computation Models and Shared Memory Interactions

The behavioral complexity of concurrent programs stems from their underlying computational models' support for parallel, interleaved, or truly concurrent execution. Formal semantics provide a precise mathematical foundation for understanding the interaction between concurrent programs and shared memory. Interleaving semantics models concurrent execution as arbitrary linear interleavings of atomic operations from multiple threads along a global timeline. While this model simplifies analysis, it may fail to capture all permissible behaviors under weak hardware memory models. More refined semantic models employ event structures or operational semantics to explicitly describe the partial order relationships between memory operations (reads, writes, updates), as well as the constraints imposed by memory barriers and synchronization primitives on the visibility and ordering of these operations.

The semantic characterization of shared memory interactions must precisely reflect the nondeterministic evolution of states. This is typically achieved through labeled transition systems, where each state encapsulates a snapshot of the global shared memory along with the local states of individual threads. The transition relation defines how a single thread’s memory operations alter the global state and may trigger changes in the state observed by other threads. For semantic characterization of weak memory models (such as x86-TSO and ARMv8), it is further necessary to introduce architectural details like store buffers and memory order rules, formally defining the core issue of "when a write becomes visible to a read." This precise semantics serves as the sole basis for subsequently determining whether a specific execution trace satisfies memory safety properties.

1.3 Correspondence Between Formal Verification Methods and Memory Safety Specifications

Formal verification aims to mathematically prove that a target system satisfies its formal specifications. Verification methods targeting memory safety are primarily divided into two categories: model checking based on automated reasoning, and theorem proving based on interactive proof. Model checking systematically explores all possible concurrent execution paths of a program through state space search or symbolic execution to determine whether any path exists that violates a given memory safety specification[2]. The corresponding specifications are typically expressed as formulas in linear temporal logic or computation tree logic, describing safety properties such as "never accessing an illegal address." Theorem proving methods, on the other hand, encode both the program and its desired safety properties within a higher-order logic system (such as Coq or Isabelle/HOL), establishing their correctness through the construction of formal proofs.

The selection and construction of appropriate verification methods directly depend on the form of expression and the complexity of the memory safety specifications. For complex specifications articulated using resource logics such as separation logic, their verification often necessitates the integration of specialized program logic reasoning systems. These systems embed rules governing the

transfer of memory resources and their permissions within their inference rules, enabling the natural derivation of resource transfers and synchronization operations between threads. Refinement and abstraction techniques bridge the gap between these methods by constructing abstract models of the program (for instance, ignoring specific data values and focusing only on pointer aliasing relationships and lock states) to reduce verification complexity, while ensuring that verification conclusions drawn on this abstract model can be concretely reasoned back to the original program. This correspondence determines the feasibility and the boundaries of completeness for the verification effort.

2. Formal Modeling Methods for Concurrent Memory Safety

2.1 Abstract State Machine Modeling for Shared Resource Access Constraints

The shared memory in concurrent programs can be viewed as a collection of shared resources accessed competitively by multiple threads. To systematically analyze its access constraints, abstract state machines provide an effective modeling framework. This framework abstracts the global program state into finite or recursively enumerable configurations. Each configuration includes the abstract values of the shared memory, the program counters of individual threads, and the states of synchronization objects used to coordinate access. State transitions are triggered by the atomic operations of each thread. These operations are defined as transformation functions on the abstract state, precisely capturing the semantics of behaviors such as reading, writing, allocation, deallocation, and lock acquisition and release[3].

Introducing the constraints of synchronization primitives and weak memory models is key to enhancing the precision of the model. Synchronization mechanisms such as locks and semaphores are modeled as special resources within the state machine. Their acquisition and release operations follow specific protocols, thereby enforcing ordering constraints within state transitions. For weak memory models, the abstract state machine must be extended to incorporate intermediate states such as store buffers and memory order labels. This enables the state transition relation to simulate the visibility delays and reordering of operations that occur in actual hardware or runtime environments. Through this modeling approach, complex concurrent interactions are reduced to analyzable paths within the state space, providing a well-defined subject for subsequent searches or reasoning when verifying memory safety properties.

2.2 Logical Expression of Concurrent Execution Traces and Memory Safety Properties

A single concurrent execution of a program can be represented as an event trace, which records memory operation events from different threads and their arrangement according to a global timeline or partial order relationship. Formal modeling requires transforming such traces into objects operable by logical formulas. Atomic propositions in linear temporal logic or computation tree logic can be defined as assertions about the state at specific positions within a trace, such as "pointer p is valid here" or "address a is locked by thread t." By quantifying over events in the trace, patterns of concurrency-related defects such as data races and atomicity violations can be precisely described.

Memory safety properties are expressed within this framework as constraints on all possible execution traces. Spatial safety properties are typically formulated as global invariants, requiring that at any point in any trace, all active pointer references must point to valid and type-matching memory regions. Temporal safety properties, on the other hand, are often expressed as response specifications, ensuring that once a memory address is deallocated, no future point in that trace will contain an access to it. For complex patterns involving the dynamic transfer of resources, it is necessary to combine predicate logic to explicitly encode the ownership transfer relationships of resources within the trace, thereby linking memory safety to the correctness of resource management protocols[4].

2.3 Resource Specification Framework for Concurrent Programs Based on Separation Logic

Separation logic provides a natural foundation for describing and reasoning about a program's exclusive access to memory resources by introducing the spatial separation operator and the concept of resource holding. Its core idea is to describe the global heap memory as a separating conjunction of multiple independent sub-heaps, where ownership of each sub-heap's resources can be allocated to a specific thread or module. In a concurrent context, separation logic is extended into concurrent separation logic. Its core rules allow parallel executing threads to reason independently about their

respective portions of resources, provided the resources they operate on are logically separate. This provides a theoretical basis for modular verification.

To construct a resource specification framework applicable to concurrent memory safety, it is necessary to embed the characterization of sharing and transfer into standard separation logic. The concept of permissions is introduced to distinguish between read-only access and exclusive write access permissions to memory regions. Synchronization operations, such as the passage of a lock, are modeled as the temporary centralization and subsequent re-distribution of resource bundles. The inference rules of this framework formally stipulate how permissions split, transfer, and merge as threads are created, interact, and are destroyed. In this way, complex memory safety protocols, such as memory management in read-write locks or the publish-subscribe pattern, can be abstracted into a series of specifications for resource state transitions. Consequently, the verification of concurrent memory safety is transformed into proving the consistency of resource specifications.

3. Construction of the Memory Safety Model and Verification Algorithms

3.1 Construction and Refinement Strategies for Safety Model Invariants

The construction of safety model invariants aims to capture global properties that a program must satisfy in any intermediate state of its concurrent execution. It serves as the core bridge connecting the concrete program with its abstract safety specification. The initial form of an invariant can be derived from synchronization operations in the source code (such as lock declarations and usage) and the calling patterns of memory management APIs, forming a preliminary set of assertions concerning resource ownership and access permissions. For concurrent programs, the key challenge lies in constructing invariants that can withstand all possible thread interleavings. This typically requires introducing auxiliary variables or ghost states to encode historical interaction information or causal dependencies among threads, thereby making implicit protocols explicit. The construction process often integrates program analysis techniques, such as deriving loop invariants for loop structures or inductively reasoning about shape properties for recursive data structures[5].

The refinement strategy enhances the strength and applicability of invariants through an iterative process. Counterexample-guided refinement serves as the primary technical approach. When a verifier discovers an execution path that violates the current invariant, the path information is used to synthesize new constraint conditions to exclude such invalid interleavings. This process may involve the refinement of predicate abstraction, where new predicates are extracted from concrete counterexamples and added to the abstract domain. It may also involve the discovery of interference constraints, which explicitly mandate that certain operations must follow a specific order to prevent data races. The endpoint of refinement is to obtain a set of invariants that are both tractable for the verification tools and sufficiently strong to imply all targeted memory safety properties. More advanced strategies incorporate interpolation-based refinement, which can automatically derive suitable new predicates from infeasible paths, thereby accelerating the convergence process.

3.2 Symbolic Execution and Abstract Interpretation Algorithms for Model Verification

Symbolic execution systematically explores the state space by symbolizing program inputs and initial shared memory values, collecting path conditions along possible execution paths and symbolically updating memory states. In a concurrent environment, this is extended to concurrent symbolic execution, which requires managing multiple parallel symbolic execution contexts and their interactions. The algorithm dynamically schedules the interleaving of symbolic instructions from different threads and utilizes a constraint solver to determine the satisfiability of various path conditions, aiming to discover whether certain combinations of thread interleavings and input values could lead to memory safety violations. The primary challenges lie in path explosion and complex constraint solving. The symbolic modeling of shared memory needs to efficiently handle constraints related to pointer aliasing and array bounds. Optimization methods include introducing partial order reduction techniques to eliminate redundant interleavings and developing efficient solving strategies tailored for specific theories (such as bit-vectors and arrays)[6].

Abstract interpretation provides an algorithmic framework for performing finite over-approximation of the state space while guaranteeing soundness. It defines an abstract domain to represent sets of memory states and designs abstract transfer functions on this domain to simulate the effects of program operations. For concurrent memory safety, the abstract domain must be capable of compactly

representing pointer validity ranges, lock holding statuses, and potential interferences between threads. The abstract interpretation algorithm computes the fixpoint of abstract states at program points iteratively, with all possible runtime states being covered by this fixpoint. By designing abstract domains with sufficient expressive power that also satisfy convergence conditions (such as domains based on separation logic or extensions of region logic), this algorithm can automatically derive memory safety invariants or prove the absence of violations. To handle complex data structures, numerical abstract domains (such as intervals or octagons) are often combined with pointer analysis domains to simultaneously reason about memory layout and data value constraints.

3.3 Component-Based Design Principles for an Extensible Verification Framework

The component-based design of an extensible verification framework emphasizes the clear separation of different logical and functional layers within the verification system. Its fundamental principles include the modularization of the specification language layer, the modeling transformation layer, the core verification algorithm layer, and the result feedback layer. The specification language layer provides a declarative syntax for expressing memory safety properties and program contracts. The modeling transformation layer is responsible for automatically translating source code and specifications into the formal model required by the underlying verifier, a process that must preserve semantic consistency. The core verification algorithm layer encapsulates pluggable verification technologies, such as symbolic execution engines or abstract interpreters. Data exchange between these layers occurs through well-defined interfaces. This layered architecture enables technological advancements or replacements within any single layer to proceed independently without necessitating a complete system redesign.

To support the verification of large-scale complex systems, the framework must adopt design principles of composability and reusability. Composability allows for the separate verification of individual program modules or threads, after which these local proofs can be combined using reasoning rules based on contracts or contextual assumptions to obtain global guarantees. Reusability is manifested in the construction of shared abstraction libraries, specification pattern libraries, and specialized verification strategy packages for specific domains (such as custom memory allocators or concurrent data structures). The framework's design must also reserve extension points for integrating new memory models, synchronization primitives, or security properties. This is typically achieved by defining a universal intermediate representation or verification condition generation interface, thereby ensuring long-term applicability amid technological evolution. The loosely coupled design among components supports flexible customization and experimentation of the verification workflow, providing targeted solutions for verification tasks across different application scenarios.

Conclusion

This study systematically elaborates a complete technical system for constructing a formal-verification-based memory safety model for concurrent programs, covering the entire chain from theoretical foundations and modeling methods to verification algorithms and framework design. By providing formal definitions and classifications of memory safety properties, along with precise semantic characterizations of concurrent computational models, the research establishes a rigorous logical starting point for subsequent modeling. The proposed abstract state machine modeling, logical expression of execution traces, and resource specification framework based on separation logic offer modular and composable descriptive tools for complex concurrent memory interactions. At the verification level, the exploration of invariant construction and refinement strategies, as well as symbolic execution and abstract interpretation algorithms, provides core algorithmic support for automated and semi-automated verification. Finally, the component-based framework design principles ensure the adaptability and extensibility of the entire verification system. Future research directions include: exploring more efficient algorithms for the automatic synthesis and refinement of invariants to reduce the verification burden; investigating verification methods for memory safety under emerging heterogeneous concurrent architectures (such as GPUs and asynchronous memory models); developing richer, more declarative domain-specific specification languages to improve engineers' productivity; and promoting the deep integration of verification tools with mainstream development environments to enhance the breadth and practicality of formal methods in industrial-grade complex systems.

References

[1] Xie, Xiaofu, Zeng, Mengqi, and Pang, Fei. "Design of a Formal Verification Method for Computer Concurrent Programs." Information Security and Communications Privacy, no. 03, 2022, pp. 54-62.
[2] Yang, Yeqian, and Dai, Hongjun. "Formal Verification of the RISC-V SBI Firmware Secure Boot Process." Journal of Computer Research and Development, pp. 1-14.
[3] Li, Yushan. Formal Modeling and Verification of Microkernel Scheduler Programs. MA thesis. University of Electronic Science and Technology of China, 2025.
[4] Wang, Junyi. Formal Modeling and Verification of Complex Data Structures. MA thesis. Anhui University, 2023.
[5] Xing, Liang, et al. "Formal Verification Technology for Safety-Critical Software Based on PROMELA Models." Journal of Northwestern Polytechnical University, vol. 40, no. 05, 2022, pp. 1180-1187.
[6] Xie, Xiaofu, Zeng, Mengqi, and Pang, Fei. "Design of a Formal Verification Method for Computer Concurrent Programs." Information Security and Communications Privacy, no. 03, 2022, pp. 54-62.