electronics

MDPI

Article

Research on Cache Coherence Protocol Verification Method Based on Model Checking

Yiqiang Zhao [1], Boning Shi [1], Qizhi Zhang [1], Yidong Yuan [1,2] and Jiaji He [1,*]

[1] School of Microelectronics, Tianjin University, Tianjin 300072, China; yq_zhao@tju.edu.cn (Y.Z.); shiboning@tju.edu.cn (B.S.); qizhi_zhang@tju.edu.cn (Q.Z.); yuanyidong@sgchip.sgcc.com.cn (Y.Y.)
[2] Beijing Zhixin Microelectronics Technology Co., Ltd., Beijing 100192, China
* Correspondence: dochejj@tju.edu.cn

Abstract: This paper analyzes the underlying logic of the processor's behavior level code. It proposes an automatic model construction and formal verification method for the cache consistency protocol with the aim of ensuring data consistency in the processor and the correctness of the cache function. The main idea of this method is to analyze the register transfer level (RTL) code directly at the module level and variable level, and extract the key modules and key variables according to the code information. Then, based on key variables, conditional behavior statements are retrieved from the code, and unnecessary statements are deleted. The model construction and simplification of related core states are completed automatically, while also simultaneously generating the attribute library to be verified, using "white list" as the construction strategy. Finally, complete cache consistency protocol verification is implemented in the model detector UPPAAL. Ultimately, this mechanism reduces the 142 state-transition path-guided global states of the cache module to be verified into 4 core functional states driven by consistency protocol implementation, effectively reducing the complexity of the formal model, and extracting 32 verification attributes into 6 verification attributes, reducing the verification time cost by 76.19%.

Keywords: multi-core processor; cache coherence protocol; attribute extraction; formal verification

Citation: Zhao, Y.; Shi, B.; Zhang, Q.; Yuan, Y.; He, J. Research on Cache Coherence Protocol Verification Method Based on Model Checking. *Electronics* **2023**, *12*, 3420. https://doi.org/10.3390/electronics12163420

Academic Editor: Yue Wu

Received: 29 June 2023
Revised: 3 August 2023
Accepted: 9 August 2023
Published: 11 August 2023

1. Introduction

1.1. Background

With the advent of the digital age, all data on human activities will be traceable. A series of analysis and processing of these data constitutes an important part of human civilization, while the computer, as a carrier of analytical data, also plays a very critical role. The application of artificial intelligence [1] and cloud computing [2] is changing rapidly, and computer technology has also undergone great changes.

The computational power of a computer system is usually increased in two ways: one is to increase the arithmetic power of a single processor by enhancing the size and process of the integrated circuit. However, as the transistor density increases, the consequent heat dissipation problems make this method of increasing the performance of a single processor no longer feasible. Another way to increase the computational power of a system is to increase the number of processors. Multiple processors are processed in parallel and placed on a single chip to increase the computational throughput of the overall system, a construction we call a multiprocessor [3]. Multi-core processor systems are widely used because of their fast computational speed and high resource utilization [4], becoming a research hotspot [5].

Compared with traditional single-core processors, the data access and data synchronization mechanisms of multiprocessors are more complex. The shared cache, as an important part of the multiprocessor, connects the processor cores to the memory and can be accessed by multiple processor cores. It plays the role of coordinating the data

and instructions of each processor core. Due to the complexity of the mechanism and the importance of its function, the multi-core shared cache system is more prone to resource conflicts and resource contamination, which can cause the processor to enter into an abnormal working state, resulting in a series of serious consequences. In addition, the cache coherence protocol plays an important stabilization role as a rule for updating and transferring data in the cache. In order to guarantee the correctness of data and the legitimacy of program execution in multi-core processor systems, the industry has started to carry out the verification of cache design at the design stage.

1.2. Research Progress and Related Work

Multi-core processor refers to the two or more processor cores integrated in a chip processor structure, also known as a single-chip multiprocessor, by the United States Stanford University first proposed [6]. The processor is the most important core component of a computer because of its sophisticated and complex structure and cumbersome manufacturing process. In order to ensure the functional completeness and safety of multi-core processors, it is necessary to verify them thoroughly.

The execution trace-based modelwas analyzed to verified multicore and real-time systems in 2022 [7]. In this paper, they present the use of model-based constraints on top of user-space and kernel traces to provide weighted analysis results. Their algorithms were applied to multiple traces showing common problems for multi-core real-time systems. The experimental results show that the algorithms can quickly identify many different types of problems with a low runtime. An integrated tool was described by Grevtsev [8] for early analysis of performance, power, physical and thermal characteristics of multicore systems. It includes cycle-accurate, transaction-level SystemC-based performance models of POWER processors and system components. Kouki proposed a rapid verification framework for developing multicore [9]. The proposed method makes it possible to verify both homogeneous and heterogeneous multi-core processors with the cache coherency mechanism and to execute multithreaded programs without full system simulation.

With the increasing scale of multicore processors in recent years, the data interaction between multicore and multithreading has become more and more complex, which makes the maintenance of data in the cache more difficult. In order to ensure the efficiency and correctness of multithreaded workloads in multicore processors, the functional completeness and information correctness of caches have become a necessary condition for the development of multicore processors. Thus, the verification of cache has gained wide attention.

Regarding the cache, as a buffer for data exchange, its data maintenance mechanism obeys the cache coherence protocol, and the cache coherence protocol has also become an important basis for ensuring the functional completeness of the cache. The cache coherence protocol was first proposed in 1991 [10], and subsequent development over the years has become a hotspot for research related to cache systems. In 2012, Shield John et al. from the University of Queensland explored two different types of asymmetric coherence strategies [11], which reduced the coherence cost of unshared data by 60% with the bus-based asymmetric coherence strategy, effectively reducing the delay due to coherent messages. In 2015, Joshi of Visvesvaraya National Institute of Technology Nagpur presented a series of techniques to improve cache performance [12], focusing on the impact of parameters such as the cache miss rate, average memory access time, execution time, energy, area, scalability, etc., on the performance of caches. In 2021, Nair Arun et al., at the Grand Forks University of North Dakota, proposed a MOESIL cache coherence protocol [13] that effectively reduces the off-target rate of the coherence protocol while maintaining comparable energy consumption. In 2022, Derebasoglu of Bilkent University proposed a software mechanism to improve the effectiveness of directory-based cache coherence [14], resulting in a 13% reduction in coherence traffic. In the same year, Khalid of the University of Jeddah gave an overview of the current problems and solutions to

cache coherence [15], describing several approaches to cache coherence protocols for use in distributed systems and analyzing which protocols are most suitable.

The traditional simulation-based verification method is time-consuming and does not achieve full function coverage, making it unsuitable for complete cache coherence protocol verification. The formal verification method, in contrast, is a mathematics-based verification method that can cover all accessible states of integrated circuits and fully ensure the system's functional correctness.

Formal hardware verification can be traced back to at least 1979. Han of Xidian University summarized the formal verification technology of hardware design that emerged at the time [16], commented on its emergence reasons and main achievements, outlined its main methods, and forecasted its future development trend. The formal verification method is divided into three parts: equivalence testing [17], theorem proving, and model checking [18,19]. The latter of these is a method for thoroughly detecting all of the accessible states and behaviors in a given reaction system or model. It is widely used in verification work because it is not only highly automated but it can also generate counterexamples to inform the verifier of its specific situation [20,21]. At the moment, model checking is primarily used for software verification, such as open-source code verification [22], the static verification of application software [23], and so on. The security verification for the system on the chip is usually the hardware verification for the processor [24]. Cache coherence protocol research primarily focuses on protocol design and optimization [12], such as the creation and execution of customizable cache coherency protocols for multi-core processors [13], cache coherence protocols for optimizing process private data access [25], and so on. In general, there have been few studies on the formal verification of the cache coherence protocol, and the description of the circuit model state is relatively simple. Because of the complexity of the processor structure, model generation not only necessitates a high detector processing capacity but the explosion of the model state also complicates verification. In a hardware formal verification effort, there is a paper which presented the modeling and formal verification of the MESI cache coherence protocol [26]. The approach is based on the developed series-parallel poset methodology. The advantage is the very low space- and time complexity of the verification algorithms. The disadvantage is the very low space- and time complexity of the verification algorithms.

For confronting complications with state explosion alongside the minimal verification efficacy that can occur with traditional verification methods, this paper will perform high-level formal modeling and will also functionally verify, through model-checking technology, these cache coherence protocols. This paper will start with the behavioral code of the processor system, analyze the key modules and critical variables of the circuit without disturbing the original code logic thinking, and build a state reduction model for the core state related to the protocol, which effectively reduces the model scale and improves the modeling efficiency. Finally, in terms of verifying attribute extraction, this paper abandons the verification idea of one thing and one discussion by analyzing the verification model of one thing and one discussion. Not only that but this paper also fully automates the work of model construction and verification tool adaptation, and verifies the cache consistency function of the processor system via positive and negative proofs. The work in this paper has two advantages in the study of verification of conformance protocols: One is the automation of the modeling and verification attribute library construction, which makes the verification of hardware a very simple task. The verifier only needs to identify the functions and circuits to be verified in order to perform the verification, and does not need to be very skilled in testing. The other is the simplification of both modeling and attribute aspects in this paper, where fewer test vectors and more focused models make verification efficient.

2. Principle

The functional verification of a multi-core shared cache system using a finite state machine is investigated in this paper. The high-level applicable verification research on

multi-core cache consistency is carried out in the research process, which is based on the multi-core shared cache structure, with the finite state machine as the modeling form and model checking as the technical method.

2.1. Multi-Core Processor

Multi-core processor refers to two or more processor cores integrated in a chip processor structure, also known as a single-chip multiprocessor. The key to the design of a multiprocessor architecture is to ensure that the interconnections and communication mechanisms between the cores are stable and correct, and that the interconnections are implemented through buses or cross switches. Different processor architectures need to choose different interconnections by weighing their complexity and scalability [27,28].

Depending on the kernel organization, multi-core shared cache systems are usually classified into multi-ported uniform cache access (UCA) and non-uniform cache access (NUCA). As shown in Figure 1, UCA has only one L2 cache, which is interconnected with the L1 cache through a bus or a cross switch, and this L2 cache is evenly shared by all processor cores. The latency of L2 cache accesses is the same for each processor core. NUCA has multiple L2 caches, and each processor core has a corresponding private L2 cache, which is directly interconnected with the processor cores and accesses the other processor cores or memory data through a bus or a cross switch.

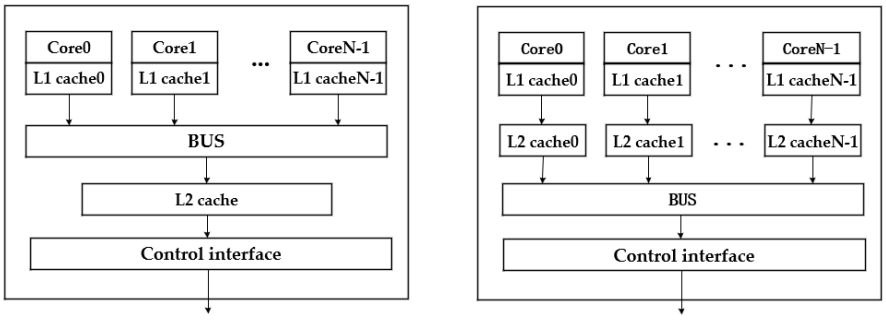

Figure 1. Multicore processor architecture.

Different levels of caches have different management areas and speeds. The L1 cache is privately owned by each processor core, and the processor core can only process the data in the first-level cache, while the distance of the cache from the processor core has a positive correlation with the data throughput speed of the cache. Then, the L1 cache is the fastest cache, and this structure can effectively allocate data to improve the resource utilization and computing efficiency of the processor system. The L1 cache is the fastest cache, and this structure can allocate data efficiently and improve the resource utilization and computational efficiency of the processor system. However, when different processor cores read, write and modify the data in a uniform location in the memory, it may cause a series of disordered competition and conflict. In order to ensure that data reading and writing are carried out in an orderly and correct manner, it is necessary to ensure that the copies of data in the caches at all levels are consistent with the data in the memory, and therefore the cache coherence protocol emerged.

2.2. Multi-Core Shared Cache Structure and Cache Coherence Protocol

Due to the complex communication mechanisms of multicore processors, cache coherence plays a crucial role in maintaining coordination. The essence of a cache coherence protocol is to specify an additional set of rules for cache access and data updates throughout the system [29]. For ease of exposition, the discussion in this subsection is all about the L2 non-uniform cache system, in which the private L1 cache and the shared L2 cache are interconnected by a bus in this system architecture. In the processor system, assume that there is a variable a with a value of 0. Both processor core 1 and processor core 3 acquire the value of variable a and generate the corresponding copy, respectively. At this time,

processor core 1 modifies the content of the copy of variable *a* to 1. Before this copy value is written back, the content of the copy of variable *a* in processor core 3 is still 0, so the value of variable *a* is in conflict. When the private caches of two processor cores fetch the same variable, if additional consistency rules are not set, it may lead to incorrect execution results.

In order to ensure that the data in each cache are consistent while still maintaining high CPU processing speed, current processors use two types of cache coherency protocol implementations: the bus listening protocol [30] and the directory-based protocol [31]. Among them, the one using bus interconnection is the bus listening protocol; the interconnection using L2 cache directory records is the directory-based protocol. And the most mainstream cache coherence protocol is the MESI protocol. The protocol is named after the possible states of the cache line, and the cache lines are coordinated to interconnect with each other in different states to ensure the correctness of the data [32].

As shown in Figure 2, if *a* wants to read this cache line, it needs to issue a GETE or GETS request to receive permission to read the latest valid data, and after the request passes, the state of this cache line will be converted to the E state or the S state, and then the processor core can read the contents of the cache line directly; if the processor core corresponding to this cache line wants to modify this cache line, it needs to issue a GETM request to get the permission to write, and after the request passes, the state of this cache line is changed to the E state or the S state. If the processor core corresponding to the cache line wants to modify the cache line, it needs to issue a GETM request to receive the write permission, and after the request is passed, the state of the cache line will be converted to the M state, and then the processor core can modify its content.

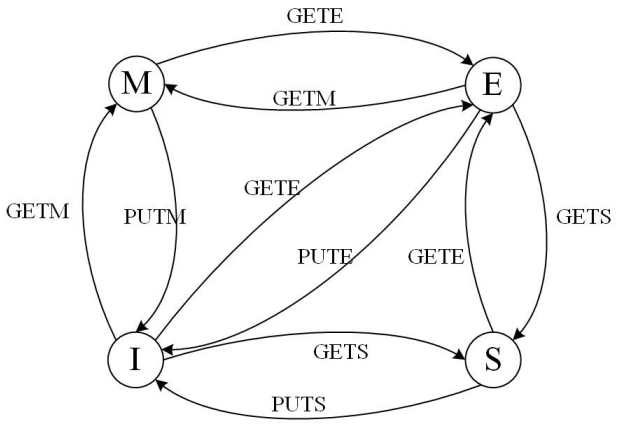

Figure 2. MESI cache coherence protocol.

If a L1 cache holds a cache line in state E, the corresponding processor core can read the contents of the cache line directly. If the processor core corresponding to this cache wants to modify the cache line, it needs to issue a GETM request to obtain write permission. After the request passes, the state of the cache line is converted to the M state, and the processor core can modify its content arbitrarily.

If a L1 cache holds a cache line in state S, the corresponding processor core can read the contents of the cache line directly. If the processor core corresponding to this cache wants to modify the cache line, it needs to issue a GETE request to receive the write permission first. After the request is passed, the state of the cache line will be changed to state E to prepare for the modification action of the processor core, and at the same time, other caches holding the same copy of the data will receive a PUTS command to change the state of their corresponding cache line to state I.

If a L1 cache holds a cache line in state M, the corresponding processor core can modify the contents of the cache line arbitrarily, and the contents of the cache line are unique. If the processor core corresponding to this cache wants to save the cache line, it needs to issue a GETE request to receive the write-back permission. After the request passes, the L1 cache reports upwards so that the contents of the L2 cache and the memory are synchronized

with it, and the state of the cache line is converted to the E state so that other processor cores are able to read the latest contents of the cache line. If the processor core corresponding to this cache does not want to save the cache line, it needs to issue a PUTM request to receive the privilege to abort the change, and after the request is passed, the state of the cache line will be changed to state I, and the contents of the cache line will be invalidated.

2.3. Model Checking Verification Principle

Model checking is a verification method that detects all of the system model's behaviors. Model checking is not only more automated than other verification methods but it can also traverse all of the model's reachable states, resulting in comparatively more complete system verification.

Model checking consists primarily of three steps: modeling, attribute writing, and verification. Modeling is the process of transforming a system to be verified into a formal model that detection tools can recognize. In model checking, attributes are used to declare the properties that the verifier wants to prove. Natural language attributes, in general, must be expressed using logical languages that are compatible with model-checking tools. During the verification process, all of the written attributes and the model to be verified are entered into the model-checking tool, and their passability is judged one by one: the representative meets the verification attribute, failure to pass indicates that this verification attribute is not met, and a counterexample is generated for the verifier to analyze the cause of the error.

Model checking has some limitations when dealing with large-scale systems. Most importantly, a 'state explosion' may occur during modeling. This state refers to the system model's possible situation, whereas the model to be detected is an exhaustive list of the system's possible state space. The corresponding state explosion is a phenomenon in which the state quantity within the complete system undergoes exponential growth in correlation with the number of system units. The larger the circuit scale corresponding to the digital integrated circuit, the more complex the structure. An in-depth formal analysis of a multi-core shared cache system, which contains a large number of logic units, control units, memory, and bus modules, will inevitably encounter the problem of state-space explosion. As a result, while modeling the system, we must overcome the limitations of state space and simplify it in order to alleviate the problem of state explosion.

3. Method

3.1. Circuit State Machine

The circuit state machine, also called the finite state machine, is a state model description of a temporal circuit [33]. Unlike combinational logic, a circuit whose output is only related to the inputs, for temporal logic, its output is not only related to the inputs but is also affected by the previous state. As shown in Figure 3, a combinational logic circuit and a memory circuit together form a temporal logic circuit, and the output of the combinational logic circuit is partially fed back to the input of the combinational logic circuit through the memory circuit, and again through the combinational logic circuit, which will have an effect on the output of the combinational logic circuit.

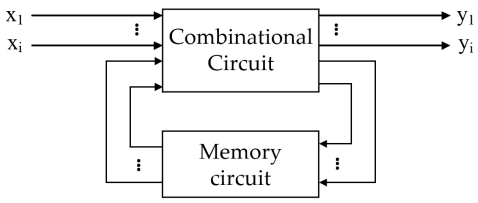

Figure 3. MESI cache coherence protocol.

For today's digital integrated circuits, even the outputs of combinational circuits are affected by timing logic. Therefore, in practice, finite state machines can be used not only to describe timing logic circuits but also for digital circuits that contain both timing logic and

combinational logic. There are many kinds of logic representations of finite state machines, the most mainstream of which are state transfer diagrams and state transfer tables. Shown in Figure 4 and Table 1 ("-" in the table means that the state remains unchanged) are a typical finite state machine state transfer diagram and state transfer table with two ways of expression, both by the initial state, a number of intermediate states, the end of the state, as well as the state of the transfer between the states.These transfer conditions are usually control signals, and when these control signals reach a specific value of the state, the circuit will make changes to the relevant data and other actions so that the circuit enters into a new state.

Figure 4. State transfer diagram.

Table 1. State transfer table.

	Condition1	**Condition2**
State1	State2	-
State2	-	-

In the process of exploring the state description of a circuit, it can be found that the number of states of a circuit is exponentially related to the triggers. For example, if a circuit contains N flip-flops, the states of these flip-flop originals are either "0" or "1", which means that the whole circuit contains a total of 2^N states. Therefore, when analyzing large-scale integrated circuits, there is a possibility of state explosion due to limited resources, making it difficult to model and analyze the circuit. Therefore, for larger scale circuits, it is necessary to divide them into modules or subdivided functional levels.

The RTL code of circuit design, which is at a high level in the production design flow of integrated circuits, expresses the expected function of the circuit by means of behavioral statements, and is an important node to undertake the functional design and the physical implementation of the circuit. In this paper, we will carry out research on formal modeling techniques for the RTL code of circuits.

Assuming that the RTL code of a complete circuit system is denoted as V, where each of the registers with different functions is denoted as v, and the hardware implementation that connects these registers is denoted as v_{in}, Equation (1) shows the complete system code composition:

$$V := v_1 \wedge v_2 \wedge \ldots \wedge v_n \wedge v_{in} \tag{1}$$

By denoting the internal state transfer relation of a register variable as f and its internal state as m, the transfer of a single register variable is as in Equation (2):

$$v \xrightarrow{f} m \tag{2}$$

By denoting the internal state transfer relation of a register variable as f and its internal state as m, the transfer of a single register variable is as in Equation (3):

$$V \xrightarrow{F} M \tag{3}$$

The finite state machine of a circuit system is a description of the state of the entire system, and therefore, the states of all the register variables together constitute the overall state of the circuit system, as in Equation (4):

$$M := m_1 \cup m_2 \cup \ldots \cup m_n \tag{4}$$

The signal that can cause state m of a single register to shift is noted as sig, whereupon the logic cell can move to the next state m' when a particular signal reaches a particular value, as in Equation (5):

$$m \overset{sig}{\rightarrow} m' \tag{5}$$

The signal that can cause a shift in the global state of the circuit system is noted as SIG, and so SIG can be expressed as Equation (6):

$$SIG := sig_1 \cup sig_2 \cup \ldots \cup sig_n \tag{6}$$

The overall state M of the circuit system moves to the next state M' when it receives the global signal SIG as in Equation (7):

$$M \overset{SIG}{\rightarrow} M' \tag{7}$$

Taking the global state M of the circuit system as a finite state and the global signal SIG as a finite state transfer condition, the global finite state machine of the circuit system is shown in Equation (8):

$$M_1 \overset{SIG_1}{\rightarrow} M_2 \overset{SIG_2}{\rightarrow} M_3 \rightarrow \ldots \rightarrow M_n \tag{8}$$

3.2. Model Construction

In contrast to the modeling method that only analyzes register units in netlist files, the modeling method proposed in this paper can directly analyze behavior files and automatically generate formal models. This method not only preserves the circuits' original functional behavior information but also applies to combinational logic. In this paper, the logic analysis of behavior-level circuits is performed, and functional construction based on behavior-level circuits is realized, effectively reducing the modeling difficulty and improving the modeling efficiency.

In this paper, the automatic modeling process will be divided into three parts: behavior-level module information analysis, tracking effective state and transition conditions based on critical variables, and standardizing output as shown in Figure 5.

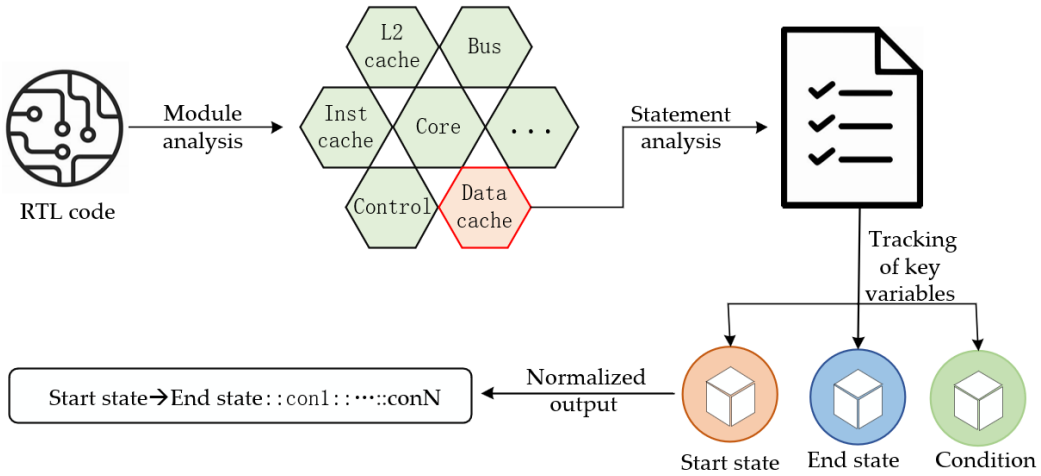

Figure 5. Automated formal modeling process.

3.2.1. RTL Code Information Analysis

For specific verification, not all register variables in the cache system circuit are necessary, so unnecessary register variables or even unnecessary modules need to be screened and removed. Due to the complexity of the IC system, this paper screens the circuit at the module level as well as the register level based on the functional information of the system, and retains only the parts related to the functional verification requirements for model construction so as to facilitate the efficiency of the subsequent verification.

In the process of high-level verification research on the function of the integrated circuit system, in addition to clarifying the functional information to be verified, it is also necessary to clarify the responsibilities of the modules and parts of the system circuit

so as to facilitate the construction of an accurate model of the system. Taking the multi-core shared cache system with uniform cache access as an example, when exploring the functional verification of the cache coherence protocol, it is necessary to firstly clarify in which module the hardware implementation of the cache coherence protocol is embodied, assuming that the state of the cache module in the circuit system is recorded as M_{cache}, and the states of other irrelevant modules are recorded as M_1, M_2, M_3 ……, …, then the state of the whole circuit system is as in Equation (9):

$$M := M_{cache} \cup M_1 \cup M_2 \cup \ldots \cup M_n \tag{9}$$

In the cache module, there exists again the coherence protocol-related specific register (usually a concatenation of several register units) state, which is noted as $m_{corhenrency}$, and the other irrelevant register states are noted as m_1, m_2, m_3,…, then the state of the whole cache module is as in Equation (10):

$$M_{cache} := m_{corhenrency} \cup m_1 \cup m_2 \cup \ldots \cup m_n \tag{10}$$

It is clear that neither the state of irrelevant modules nor the state of irrelevant registers affects the exploration of coherence protocols, and so analyzing the state of irrelevant registers would result in a certain degree of wasted resources. Therefore, for the verification of a specific function, only the state of the relevant registers needs to be modeled, and so for the verification of the cache coherence protocol function, only the relevant register state $m_{corhenrency}$ needs to be modeled.

In timing logic circuits, it is necessary to specify the names S1 and S2 of the relevant register variables before and after the timing, where S1 represents the state before the register variable is assigned, i.e., the previous state of the finite state machine, and S2 represents the state after the register variable is assigned, i.e., the state of the finite state machine after a state jump occurs due to the input of a certain condition. Among them, variable S1 and variable S2 are not necessarily specific to a particular register but can be a combination or partially intercepted combination of multiple variables in the circuit; however, they need to cover the full functional expression, and are the basis for the subsequent construction of functionally relevant state models.

3.2.2. Behavioral Tracking of Key Variables

Second, key variables must be tracked. In RTL circuits, variable S1 and variable S2 are noted as key variables affecting the consistency of the multi-core shared cache, which will be traced in the following for circuit behavior. The main idea of this part is to filter the states and conditions involving the key variables, and complete the construction of the formal model using only the S1 and S2 variable names and initial codes as inputs. As shown in Figure 6, it is necessary to first intercept the portion containing the occurrence of the assigned statement of variable S2 as a valid code segment, and then carry out the aggregation and collation of the condition information with the circuit behavior, and finally generate a number of sets of sufficient conditions oriented to the transfer of the state, the specific process being as follows.

Variable S1 is used as a possible start state in a conditional statement and variable S2 is used as a possible end state in an assignment statement. Firstly, the range of values of the two variables is determined. Secondly, the different states associated with different assignments of the variables are defined. Thirdly, the global behavior is then traced according to the key state. Finally, the start state, end state and transfer conditions for the state transfer are summarized in the form of a state machine.

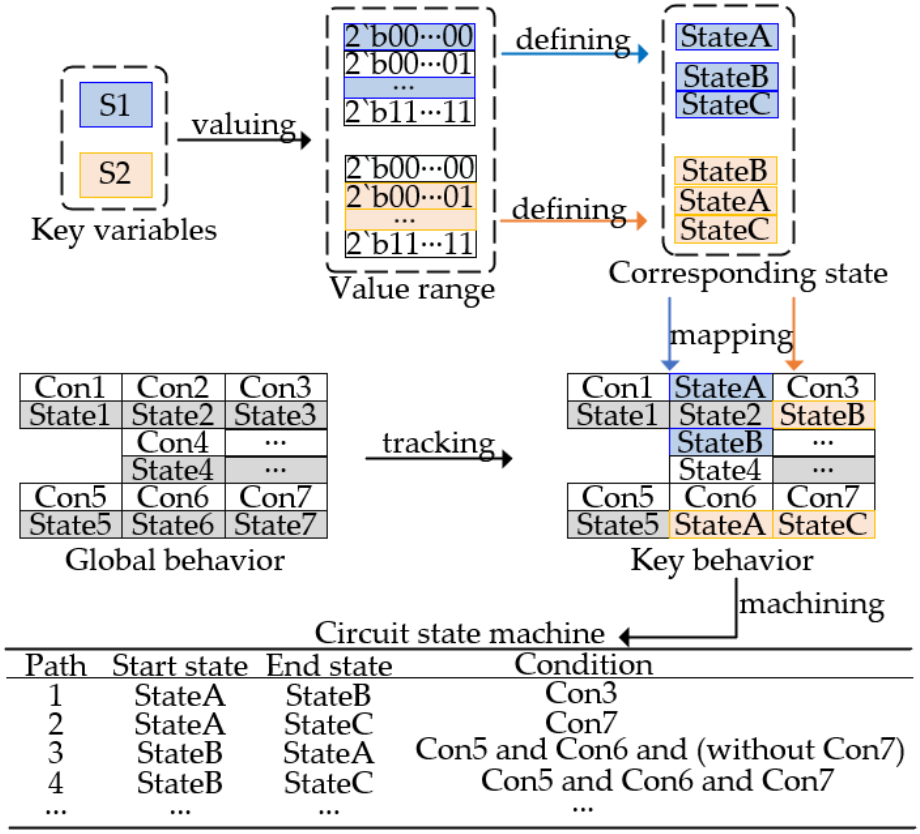

Figure 6. Tracking of key variables.

3.2.3. Normalized Output

Normalized output is the final step in formal modeling. Following the completion of the preceding operations, the system obtains several S2 assignment statements and their corresponding sufficient conversion condition statements, which must then be normalized by a finite state machine. The finite state machine is made up of three parts: the initial state, the end state, and the meeting of specific conditions that can result in transformation. This paper's output format is as follows:

$$S1 - - > S2 :: C1 :: C2 :: \ldots :: Cn \tag{11}$$

where S1 represents the initial state, such as the condition before the logical jump; S2 represents the end state, including the condition after the logical jump; and C1, C2, . . ., Cn represent transformation conditions, and all conditions are in a parallel "AND" relationship. This state transition's preliminary condition is state1, the end condition is state2, and the transformation conditions are A = value2 and condition1, so the output of the state transition should be

$$state1 - - > state2 :: A == value2 :: condition1 \tag{12}$$

For any S2 assignment statement, you can obtain one or several state-transition statements as shown in the above formula, traverse all S2 assignment statements appearing in behavioral code, and output them line by line according to the format, and then you can obtain a standardized finite state machine model.

3.3. Generation of Attribute Library to Be Verified

For a formal verification system for model checking, the verifier needs to write the verification properties according to the structure of the system to be verified. The method of describing the verification properties needs to combine the model structure with the

verification topic, and at the same time, the reliability of the whole model system needs to be taken into account. In the process of writing the description of validation attributes for the system, the natural language is generally written according to the validation objective, and then the attributes described in the natural language are translated into the formal description language adapted by the validation tool by combining the specific model and validation method, and finally the formal description language is written into the validation system of the formal model so as to validate the passability of each attribute.

In the following, from the introduction of the formal description language, this section proposes an attribute extraction method based on "whitelisting" and "blacklisting" for the cache consistency protocol model, which solves the problem of "one issue" of the current formal validation attributes and effectively improves the validation efficiency.

The verification attribute is a direct factor in determining whether the verification system can meet the verification requirements. The extraction process is divided into two parts: writing natural language in accordance with the verification target and writing logical language in accordance with the natural language. The following is the basic formula for a logic language based on computation tree logic (CTL):

$$A[]Process.state1 \tag{13}$$

$$A <> Process.state1 \tag{14}$$

$$E[]Process.state1 \tag{15}$$

$$E <> Process.state1 \tag{16}$$

$$Process.state1 - - > Process.state2 \tag{17}$$

where A stands for all paths, E stands for the existence of a path, [] stands for all states on the path, < > stands for a future state, Process stands for a certain model, state1 and state2 stand for certain states in a certain model, –> stands for the logical state of "causing", and the logical language expression with the same meaning as -> is A [](Process.state1 imply A<> Process.state2).

For a cache-consistent functional verification system, the traditional approach requires traversing all the paths in the circuit model and determining the correctness of the state transfer conditions and state accessibility one by one, implying that as many paths exist in the system as there are verification attributes to write.However, the time cost of this "one-issue" verification approach is high, and the number of paths in the formal model is nearly square to the number of states in the model, making it cumbersome to write attributes for all paths in a large-scale model.

Based on the analysis of the original circuit function and the in-depth discussion of the functional model, this thesis proposes a functional verification attribute extraction method combining "white list" and "black list" guided by the verification objectives. The method starts from the three aspects of state accessibility, no deadlock and functional completeness, and creates a functional verification attribute library for the system based on the structural characteristics of the formal model.

The test of circuit reliability depends on whether the system satisfies the following three conditions after the design is completed:

1 State reachability: for every state in the formal model, there exists at least one path that enables the model to reach that state. This property is mandated by the verification system and is one of the properties that the verifier expects to pass, so it can be described as a "whitelisted" property based on the content of the property. All states in the model are reachable, corresponding to the formal language format of Equation (18), where k represents the state number, which is the same as the number of states in the model:

$$E <> Process.state_k \tag{18}$$

Assuming that the model has n states, we need to write n state accessibility attributes, perform n times of attribute verification, and use the concatenated set of all the attributes passed as the "white list" attribute library of state accessibility. However, if the "whitelisted" attribute pool of the forward test is large, it is possible to reverse test the attributes, replacing the pass test for state reachability with a fail test for state unreachability, i.e., to determine whether any state in the model is unreachable, and this attribute is the one that the verifier expects to fail. The formal language can be transferred to generate a library of "whitelisted" attributes. The corresponding formal language format is as in Equation (19):

$$A[]not\ Process.state_k \tag{19}$$

Similarly, a union set is taken for an n-th attribute description, with the difference that the formal linguistic format of such a union set of negative attribute descriptions is as in Equation (20):

$$A[]((not\ Process.state_1)\ and\ (not\ Process.state_2)\ and \ldots (not\ Process.state_n))) \tag{20}$$

So this attribute description statement can be used as a "whitelist" attribute library for state reachability, and the volume is changed from n to 1, which means that only one pass can be performed, and attribute failure means that the circuit model to be verified meets the "state reachability" criteria. The attribute does not pass, which means that the circuit model to be verified satisfies the "state reachability".

2 No deadlock: For the whole formal model system, there cannot exist any inescapable path loop or single state, so that the model cannot reach any state other than this loop path state under any condition, resulting in the circuit being trapped in a dead loop and unable to operate normally. This property is definitely expressed as a dead loop, which is not expected by the verifier, and can therefore be described as a "blacklist" property according to its content: if the formal model may enter any inescapable path loop or single state, then there is a risk that the model will enter a dead loop, and the circuit will not function properly. The function is also at some risk of being paralyzed. Because of the importance of the absence of dead cycles to the model, there is a property description Formula (21) in the CTL formal language system specifically for dead cycles:

$$A[]not\ deadlock \tag{21}$$

The verifier can include the failure of this attribute in the "blacklist" attribute library of the verification system to determine the risk of deadlock in the circuit model.

3 Functional completeness: For all validation targets of the formal model, their passes are consistent with the design goals of the circuit system, and can satisfy all relevant functions expected from the circuit. All verification attributes describing the expected functionality are expected to be passed by the verifier, and therefore, the functional verification attributes can be included in the "whitelist" attribute library. Such functional goal-oriented verification attributes are more targeted and efficient in the verification process, and unnecessary paths can be discarded. However, the state and condition descriptions in such verification attributes are usually relative to the whole model, and can be refined and generalized in the actual analysis process. As shown in the figure, suppose there are n states in the formal model, which are denoted as $s_1, s_2 \ldots s_n$, then the formal linguistic representation of the entire state machine set S of the formal model is as in Equation (22):

$$S := s_1 \wedge s_2 \wedge \ldots \wedge s_n \tag{22}$$

Noting the conditional statement in the formal model as $cons_k$, the condition of any state transfer path in the formal model satisfies Equation (23):

$$Path_{j-k} = cons_1 \cap cons_2 \cap \ldots cons_n \tag{23}$$

In formal models of real circuits, states and conditional statements are usually linked, and there may be situations where a particular class of conditions will cause the system to enter a particular state, which can be extracted inductively from the validation attributes, which are classified into a total of three, and will be explored in the following paper, one by one.

① Initial state induction: There may exist certain sets of states in the formal model, and there exists a certain condition that enables the model to enter a certain state from a set of states. For example, there exists a set of states denoted as S1 , which contains the sub-states s1 and s2, respectively, and so the formal linguistic expression for the set of states S1 is as in Equation (24):

$$S_1 := s_1 \wedge s_2 \tag{24}$$

At this time, there exists a transfer condition denoted as $cons_j$, and a state other than the state set S1 is denoted as s_j. When the system model is currently in any one of the states in the state set S1 and satisfies the condition, the model will enter state s_j. For this case, this paper will focus on the initial state and the end state of the validation attributes involved in the generalization of the results of the generalization of the formula (refeqq), written into the "white list" attribute library:

$$(s_1 \ or \ s_2) \ and \ cons_j - - > s_j \tag{25}$$

② Transfer condition induction: There exist certain sets of state transfer conditions in a formal model, any one of which can enable the model to move from one state to another. For example, there exists a set of state transfer conditions denoted as CON_1, which contains the sub-state transfer conditions $cons_1$ and $cons_2$, respectively, whereupon the formal linguistic expression for the set of state transfer conditions CON_1 is expressed as Equation (26):

$$CON_1 = cons_1 \cup cons_2 \tag{26}$$

At this point, the initial state of the state transfer condition connection is noted as s_5, and the end state is noted as s_6. When the system model is currently in state s_5 and satisfies any one of the conditions in the set of state transfer conditions, the model will enter state s_6. For this case, we will summarize the state transfer conditions of the involved validation attributes and write them into the "whitelist" attribute library, and the formal language expression of the summarization result is as in Equation (27):

$$s_5 \ and \ (cons_1 \ and \ cons_2) - - > s_6 \tag{27}$$

The above two generalizable cases are not mutually exclusive; in the process of extracting attributes from the actual validation model, there may be cases where attributes are applicable to multiple extraction methods, i.e., the initial state, the end state and the state transfer condition can be generalized at the same time, and the specifics need to be analyzed according to the actual situation of the model.

3.4. The Whole Process Automation

As shown in Figure 7, this paper generates a tool-recognizable verification model based on behavioral hardware language in an automated form and constructs a functional model-checking interface.

This paper obtained the standard state transition file during the modeling process in Section 2.1, which can then be converted into the model language. A complete model is divided into four parts in the visual model tool: parameter declaration, position declaration, transition condition declaration, and attribute declaration. The parameter declaration section declares all conditional statements' judgment signals and writes them into the < declaration > section. State positions are planned in the position declaration section as a square matrix based on the number of states and written into < location >. For example, eight states are placed in a 3×3 square matrix. Each state is connected in a directed manner in the transition condition declaration part, and the corresponding conditions are declared

and written into the < transition > part. The attribute declaration section is where you write the attribute to be verified into the < query > section in a logical language.

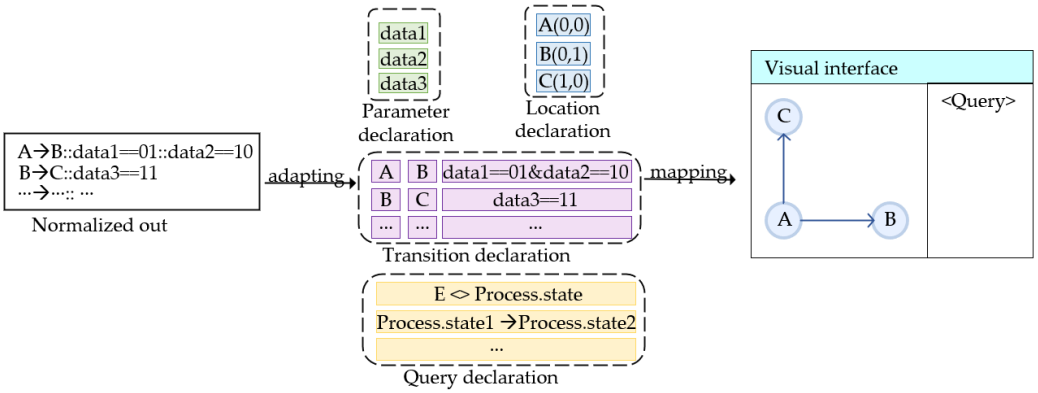

Figure 7. The whole process automation.

4. Experimental Results and Analysis

This paper uses the multi-core processor system designed by Zhai Shaomin and others at Xidian University as the benchmark circuit [34], which is a processor system with four processor cores and a secondary cache, to validate how effective the suggested process truly is. The processor core can be understood as the system's central processing unit, responsible for data processing and operation. The first-level cache is divided into two parts: data cache and instruction cache, which are used to temporarily store the data that the processor core is processing. The secondary cache is directly connected to the memory, which is closer to the memory and has more storage space but a slower data reading speed. Memory is the primary component used to store data.

In terms of verification tools, this paper uses UPPAAL [35], an automated verification tool, as the model-checking verification object. UPPAAL is a model-checking tool based on time automata developed jointly by Uppsala University's School of Information Technology and Aalborg University's School of Computing Science. This tool can not only automatically verify the system but it also has a clear and understandable user graphical interface and can generate a visual state transition process diagram while verifying the system thoroughly.

The editor, simulator, and verifier are the three components of UPPAAL. The editor, for example, is used to edit the model, from which an intuitive state transition diagram can be generated. The simulator can simulate how the state transition model jumps, and it can also create the counterexample diagram. The verifier is the component that produces verification results and writes verification attributes. The verifier can judge whether the verification attributes pass one by one, and if the attributes fail, a counterexample is generated in the simulator.

4.1. Model Construction Results Based on Cache Consistency

The variables responsible for coordinating the consistency of multi-core cache are in the data cache in the processor structure adopted in this paper, which is not only responsible for data pre-writing, data cache, and other contents but also has the content of coordinating cache consistency. Its signal transmission and reception are usually accompanied by data reading and writing or storage, so this paper chooses to construct a formal model for the data cache part.

The data cache is modeled globally without discrimination using the method described above, and there are 142 states. There are nearly 142^{142} state transition routes among 142 states for the function of cache consistency. Every attribute determination in the process of state accessibility-oriented function verification must traverse a model with over 20,000 possible routes. The MESI cache coherence protocol only cares about whether the four states of MESI can jump correctly after receiving the instruction. As a result, the number of state transition routes is reduced from 142^{142} to 32. The behavior of related modules

reveals that MESI corresponds to a pair of two-bit variables: state_in and state_out, with value ranges of 00, 01, 10, and 11, representing the four states I, E, S, and M, respectively. Among these are the variables state_in, which depicts the state before the logical action jumps, and state_out, which represents the state after the jump. We only need to calibrate and track these two variables to finish the full-automatic formal model construction with cache consistency as the function orientation (see Figure 8 for a schematic diagram of model points).

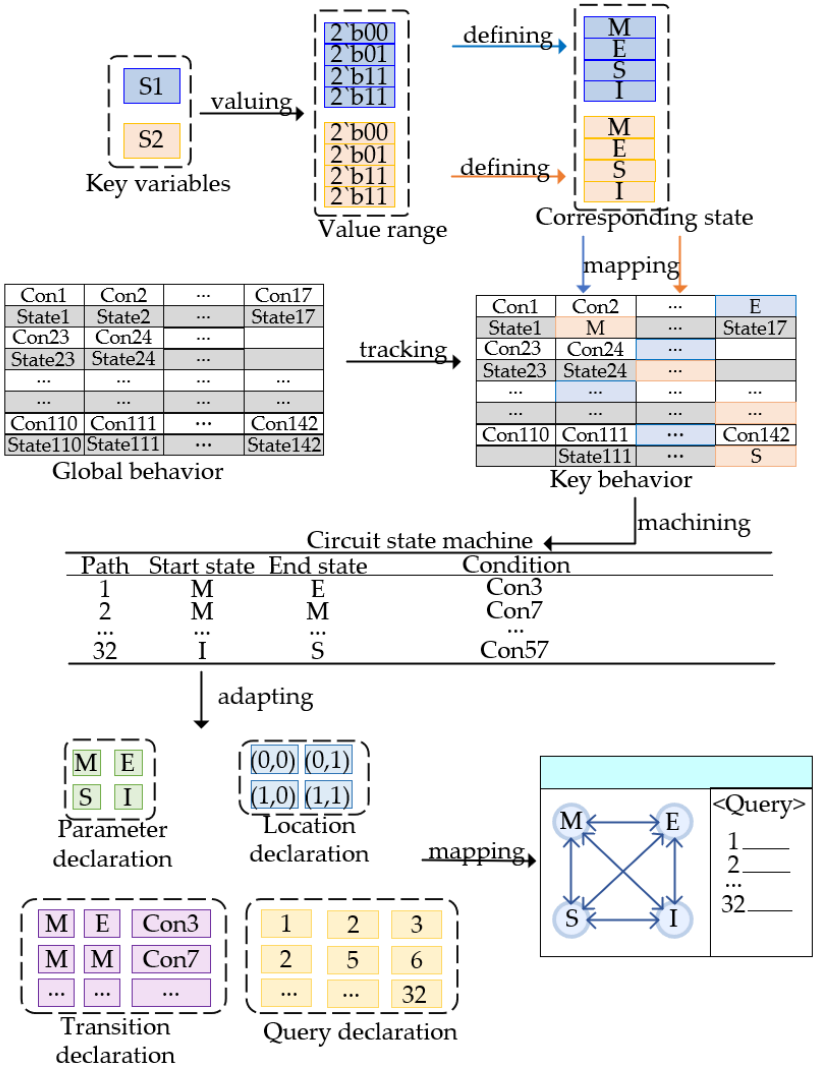

Figure 8. Cache consistency model.

4.2. Verification Result of MESI Function Based on White List

The four states in Figure 4 correspond to the four states of MESI of the cache coherence protocol, respectively. In order to verify whether its functionality meets the cache coherence protocol, it is necessary to first explore the natural language attributes of the cache coherence protocol. Table 2 illustrates the conversion conditions of cache coherence protocol in natural language by verifying the attribute "white list" :

Table 2. "White list" attribute library in natural language.

Initial State	End State	Transfer Condition
I or E	E	Local processor core read operation
/ [1]	S	Check the same data reading operation with other processors locally.
S	I	Other processor core writes operations
/	M	Local processor core writes operation
M	E	Local processor core writeback operation
/	I	Empty operation

[1] "/" in the table stands for any state.

Table 2 contains 32 attributes to be verified, which represent the directed transition and transition conditions among all cache states. However, there is a problem with some specific operations, in that a single event is discussed numerous times. For example, regardless of the initial state of the cache, when the processor core is emptying the cache, the end state of the cache will become the I state. As a result, this paper analyzes the typical characteristics of the verification target, uses the end state as the fixed result orientation, extracts the initial conditions with repeated transfer conditions, and finally generates the attribute library to be verified with a volume of 6, reducing the verification process time cost.

After completing the analysis and conversion from natural language to logical language, it must be entered into the verifier and verified. Passing indicates that the circuit satisfies the verification attribute, while failing indicates that the verification attribute is not satisfied. The verification used in this paper is both positive and negative, in that the forward verification determines whether the function is correct, and the reverse verification generates counterexamples and displays the function execution process in the simulator interface. Table 3 shows the positive and negative verification attributes as well as the verification results:

Table 3. Verification results.

Initial State	End State	Verification Result	Time (s)
I or E	E	PASS	0.003
/	S	PASS	0.001
S	I	PASS	0.002
/	M	PASS	0.002
M	E	PASS	0.002
/	I	PASS	0.001
No deadlock	A[] not deadlock	Pass	0.003
State reachability	A[] ((not P1.M) and (not P1.E) and (not P1.S) and (not P1. I))	NOT PASS	0.001

The pass/fail results all meet the expectations, proving that the cache design satisfies state reachability, no deadlock, and consistency completeness. In the validation results of the circuit cache consistency completeness, it can be found that all the consistency protocol judgment validations are passed. This means that the cache is designed to transfer data with complete rules, which ensures that the system is well organized. In the deadlock exploration of this circuit, we obtain the conclusion that no deadlock is found. Due to the comprehensive formal verification, we can be confident that the circuit will not fall into a deadlock state. During the verification of the state reachability of the circuit, we perform an iterative verification of the possible states. The query is also simplified using a form of reverse validation, and the validation results show NOT PASS as expected. Thus, we can learn that the consistency protocol of the cache circuit design is complete and can serve as a stabilization.

By counting the verification time, it can be learned that the whole process of verifying the functionality of the conformance protocol of the model takes a total of 15 ms. In total,

32 functionality-complete attributes generated by the model are returned to the beginning, along with 4 state-reachable attributes, and they are verified one by one. The validation time is then counted and totaled as 6 ms as shown in Figure 9. Attribute extraction enables the validation process to save a total of 76.19% of the time.

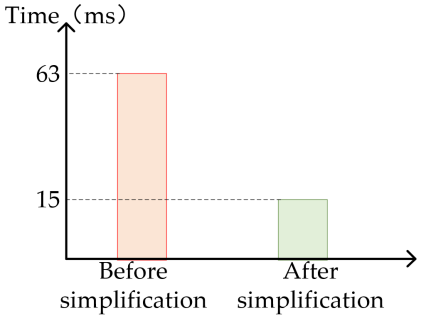

Figure 9. Comparison of time before and after simplification.

There are two reasons why the validation is so efficient: one is that we simplified the model, which makes the tool need to take fewer paths in the validation process; the other is that we simplified the validation attributes, which means that a single validation statement can validate multiple attributes at the same time, which effectively saves the validator's time cost.

5. Conclusions

This paper proposes an automatic multi-core cache consistency verification method based on behavior-level code to address the low efficiency of cache consistency verification. By analyzing the code's behavior characteristics, this method can automatically build a formal model of cache consistency and generate the model interface and verification interface in UPPAAL, a verification tool. This paper's investigational findings demonstrate how this process reduces the number of states that need to be analyzed by 97%, completes the automatic construction of formal models with low resource overhead, effectively optimizes the extraction of verification attributes, reduces the time cost of the verification attribute library by 76.19%, and performs accurate and complete high-level cache consistency verification by generating counterexamples.

The work in this paper has two advantages in the study of the verification of conformance protocols: one is the automation of the modeling and verification attribute library construction, which makes the verification of hardware a very simple task. The verifier only needs to identify the functions and circuits to be verified in order to perform the verification, and does not need to be very skilled in testing. The other is the simplification of both modeling and attribute aspects in this paper, where fewer test vectors and more focused models make verification efficient.

Author Contributions: Conceptualization, B.S. and Y.Z.; methodology, Y.Y., J.H. and B.S.; software, B.S. and Q.Z.; investigation, B.S.; writing—original draft preparation, B.S., Y.Z. and J.H.; writing—review and editing, B.S., Q.Z. and Y.Z.; supervision, Y.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded in part by the National Key Research and Development Plan of China (Grant No. 2021YFB3100901) and the National Natural Science Foundation of Tianjin City (22JCQNJC00970).

Data Availability Statement: The data that support the findings of this study are available on request from the corresponding author. The data are not publicly available due to privacy or ethical restrictions.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study, the collection, analyses, interpretation of data, the writing of the manuscript,or in the decision to publish the results.

References

1. Harika, J.; Baleeshwar, P.; Navya, K.; Shanmugasundaram, H. A review on artificial intelligence with deep human reasoning. In Proceedings of the 2022 International Conference on Applied Artificial Intelligence and Computing (ICAAIC), Salem, India, 9–11 May 2022 ; pp. 81–84.
2. Bhari, S.; Quraishi, S.J. Blockchain and cloud computing-a review. In Proceedings of the 2022 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COM-IT-CON), Faridabad, India, 26–27 May 2022; Volume 1, pp. 766–770.
3. Rahman, M.M. Process synchronization in multiprocessor and multi-core processor. In Proceedings of the 2012 International Conference on Informatics, Electronics & Vision (ICIEV), Dhaka, Bangladesh, 18–19 May 2012; pp. 554–559.
4. Kostadinov, A.N.; Kouzaev, G.A. A novel processor for artificial intelligence acceleration. *WSEAS Trans. Circuits Syst.* **2022**. Available online: https://api.semanticscholar.org/CorpusID:250237892 (accessed on 15 October 2022). [CrossRef]
5. Chen, W.-H.; Dou, C.; Li, K.-X.; Lin, W.-Y.; Li, P.-Y.; Huang, J.-H.; Wang, J.-H.; Wei, W.-C.; Xue, C.-X.; Chiu, Y.-C.; et al. Cmos-integrated memristive non-volatile computing-in-memory for ai edge processors. *Nat. Electron.* **2019**, *2*, 420–428. Available online: https://api.semanticscholar.org/CorpusID:202771829 (accessed on 14 October 2022) [CrossRef]
6. Hammond, L.; Hubbert, B.; Siu, M.; Prabhu, M.; Chen, M.; Olukolun, K. The stanford hydra cmp. *IEEE Micro* **2000**, *20*, 71–84. [CrossRef]
7. Beamonte, R.; Ezzati-Jivan, N.; Dagenais, M.R. Execution trace-based model verification to analyze multicore and real-time systems. *Concurr. Comput. Pract. Exp.* **2022**, *34*, e6974. Available online: https://api.semanticscholar.org/CorpusID:248591680 (accessed on 4 January 2023) [CrossRef]
8. Grevtsev, N.A.; Chibisov, P.A. Multicore processor models verification in the early stages. *Probl. Adv. Micro- Nanoelectron. Syst. Dev.* **2019**. Available online: https://api.semanticscholar.org/CorpusID:201881161 (accessed on 7 October 2022). [CrossRef]
9. Kayamuro, K.; Sasaki, T.; Fukazawa, Y.; Kondo, T. A rapid verification framework for developing multi-core processor. In Proceedings of the 2016 Fourth International Symposium on Computing and Networking (CANDAR), Hiroshima, Japan, 22–25 November 2016; pp. 388–394.
10. Agarwal, A.; Simoni, R.; Hennessy, J.; Horowitz, M. An evaluation of directory schemes for cache coherence. *ACM Sigarch Comput. Archit. News* **1988**, *16*, 280–298. [CrossRef]
11. Shield, J.; Diguet, J.-P.; Gogniat, G. Asymmetric cache coherency: Policy modifications to improve multicore performance. *ACM Trans. Reconfigurable Technol. Syst.* **2012**, *5*, 3 [CrossRef]
12. Joshi, A.D.; Ramasubramanian, N. Comparison of significant issues in multicore cache coherence . In Proceedings of the 2015 International Conference on Green Computing and Internet of Things (ICGCIOT), Greater Noida, India, 8–10 October 2015; pp. 108–112.
13. Nair, A.S.; Pai, A.V.; Raveendran, B.K.; Patil, G. Moesil: A cache coherency protocol for locked mixed criticality l1 data cache . In Proceedings of the 2021 IEEE/ACM 25th International Symposium on Distributed Simulation and Real Time Applications (DS-RT 2021), Valencia, Spain, 27–29 September 2021; Cecilia, J., Martinez, F., Eds.; IEEE ACM International Symposium on Distributed Simulation and Real-Time Applications; IEEE: Piscataway Township, NJ, USA, 2021.
14. Derebasoglu, E.; Kadayif, I.; Ozturk, O. Coherency traffic reduction in manycore systems. In Proceedings of the 2022 25th Euromicro Conference on Digital System Design (DSD), Maspalomas, Spain, 31 August 2022–2 September 2022; Fabelo, H., Ortega, S., Skavhaug, A., Eds.; EUROMICRO Conference Proceedings; IEEE: Piscataway Township, NJ, USA, 2022; pp. 262–267.
15. Alkhamisi, K. Cache coherence issues and solution: A review. *Int. J. Inf. Syst. Comput. Technol.* **2022**, *1*, 2. [CrossRef]
16. Carter, W.; Joyner, W.; Brand, D. Symbolic simulation for correct machine design. In Proceedings of the 16th Design Automation Conference, San Diego, CA, USA, 25–27 June 1979; pp. 280–286.
17. Singh, A. Equivalence checking of non-binary combinational netlists. In Proceedings of the 2022 35th International Conference on VLSI Design and 2022 21st International Conference on Embedded Systems (VLSID), Bangalore, India, 26 February–2 March 2022; pp. 22–27.
18. Borek, M.; Stenzel, K.; Katkalov, K.; Reif, W. Abstracting security-critical applications for model checking in a model-driven approach. In Proceedings of the 2015 6th IEEE International Conference on Software Engineering and Service Science (ICSESS), Beijing, China, 23–25 September 2015; pp. 11–14.
19. Liu, Y.; He, C. A heuristics-based incremental probabilistic model checking at runtime. In Proceedings of the 2020 IEEE 11th International Conference on Software Engineering and Service Science (ICSESS), Beijing, China, 16–18 October 2020; pp. 355–358.
20. Adesina, O.; Lethbridge, T.C.; Somé, S. Optimizing hierarchical, concurrent state machines in umple for model checking. In Proceedings of the 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), Munich, Germany, 15–20 September 2019; pp. 524–532.
21. Zhu, W. Model checking for alphacode-generated programs. In Proceedings of the 2022 7th International Conference on Intelligent Computing and Signal Processing (ICSP), Xi'an, China, 15–17 April 2022; pp. 794–798.
22. Xin, L.; Wandong, C. A program vulnerabilities detection frame by static code analysis and model checking. In Proceedings of the 2011 IEEE 3rd International Conference on Communication Software and Networks, Xi'an, China, 27–29 May 2011; pp. 130–134.
23. Zhu, W.; Feng, P.; Deng, M. An approximate ctl model checking approach. In Proceedings of the 2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS), Beijing, China, 18–20 October 2019; pp. 646–648.

24. Gholami, S.; Sarjoughian, H.S. Unified property evaluations of constrained-devs models for simulation and model checking. In Proceedings of the 2021 Annual Modeling and Simulation Conference (ANNSIM), Fairfax, VA, USA, 19–22 July 2021; pp. 1–12.
25. Dai, W.; Chen, L.; Wu, A.; Ali, M.L. Dasc: A privacy-protected data access system with cache mechanism for smartphones. In Proceedings of the 2020 29th Wireless and Optical Communications Conference (WOCC), Newark, NJ, USA, 1–2 May 2020; pp. 1–6.
26. Ivanov, L.; Nunna, R. Modeling and verification of cache coherence protocols. In Proceedings of the ISCAS 2001. The 2001 IEEE International Symposium on Circuits and Systems (Cat. No.01CH37196), Sydney, NSW, Australia, 6–9 May 2001; Volume 5, pp. 129–132.
27. Jadon, S.; Yadav, R.S. Multicore processor: Internal structure, architecture, issues, challenges, scheduling strategies and performance. In Proceedings of the 2016 11th International Conference on Industrial and Information Systems (ICIIS), Roorkee, India, 3–4 December 2016; pp. 381–386.
28. Kazempour, V.; Fedorova, A.; Alagheband, P. Performance implications of cache affinity on multicore processors. In *Euro-Par 2008 Parallel Processing, Proceedings of the 14th International Euro-Par Conference, Las Palmas de Gran Canaria, Spain, 26–29 August 2008*; Luque, E., Margalef, T., Benitez, D., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2008; Volume 5168, pp. 151–161.
29. Al-Waisi, Z.; Agyeman, M.O. An overview of on-chip cache coherence protocols. In Proceedings of the 2017 Intelligent Systems Conference (IntelliSys), London, UK, 7–8 September 2017; pp. 304–309.
30. Sandell, M.; Raza, U.; Uchida, D. Solicit: Synchronous listen, code, and transmit protocol for wireless control applications. *IEEE Syst. J.* **2023**, 1–12. [CrossRef]
31. Ahmed, R.E.; Dhodhi, M.K. Directory-based cache coherence protocol for power-aware chip-multiprocessors. In Proceedings of the 2011 24th Canadian Conference on Electrical and Computer Engineering (CCECE), Niagara Falls, ON, Canada, 8–11 May 2011; pp. 001 036–001 039.
32. Sanchez, E.; SonzaReorda, M. On the functional test of mesi controllers. In Proceedings of the 2011 12th Latin American Test Workshop (LATW), Beach of Porto de Galinhas, Brazil, 27–30 March 2011; pp. 1–6.
33. Galan, P. Finite-state machine for embedded systems. *Control Eng.* **2021**, *68*, 27–30.
34. Shaomin, Z. Ring_network-based-multicore-. 2016. Available online: https://github.com/zhaishaomin/ring_network-based-multicore- (accessed on 7 Februaryr 2022).
35. Larsen, K.; Pettersson, P.; Yi, W. Compositional and symbolic model-checking of real-time systems. In Proceedings of the Proceedings 16th IEEE Real-Time Systems Symposium, Pisa, Italy, 5–7 December 1995; pp. 76–87.