electronics

MDPI

Article

An Approach to the State Explosion Problem: SOPC Case Study

Shan Zhou, Jinbo Wang *, Panpan Xue, Xiangyang Wang and Lu Kong

Technology and Engineering Center for Space Utilization, Chinese Academy of Sciences, Beijing 100094, China; zhoushan@csu.ac.cn (S.Z.); xuepanpan@csu.ac.cn (P.X.); wangxiangyang@csu.ac.cn (X.W.); konglu@csu.ac.cn (L.K.)

* Correspondence: wangjinbo@csu.ac.cn

Abstract: The system on a programmable chip (SOPC) architecture is better than traditional central processing unit (CPU) + field-programmable gate array (FPGA) architecture. It forms an efficient coupling between processor software and programmable logic through an on-chip high-speed bus. The SOPC architecture is resource-rich and highly customizable. At the same time, it combines low power consumption and high performance, making it popular in the field of high reliability and other new industrial fields. The SOPC architecture system is complex and integrates multiple forms of intellectual property (IP). Because of this, the traditional dynamic test and the static test cannot meet the requirements for test depth. To solve the problem of verification depth, we should introduce formal verification. But there are some types of IP forms that formal tools cannot recognize. These include black box IP, encrypted IP, and netlist IP in the SOPC model. Also, the state space explosion caused by the huge scale of the SOPC model cannot be formally verified. In this paper, we propose a modeling method using SOPC architecture. The model solves the problem of formal tools not recognizing multi-form IPs. To compress the state space, we propose reducing SOPC variables and branch relationships based on verification properties. Then, we conduct a property verification experiment on the reduced SOPC model. The experiment result shows that the model can significantly reduce the verification time.

Keywords: SOPC; state explosion problem; model checking; property verification

Citation: Zhou, S.; Wang, J.; Xue, P.; Wang, X.; Kong, L. An Approach to the State Explosion Problem: SOPC Case Study. *Electronics* **2023**, *12*, 4987. https://doi.org/10.3390/electronics12244987

Academic Editors: Shibo He, Huan Zhou, Victor C. M. Leung, Fangyuan Xing and Lei Yang

Received: 9 October 2023
Revised: 27 November 2023
Accepted: 28 November 2023
Published: 13 December 2023

Correction Statement: This article has been republished with a minor change. The change does not affect the scientific content of the article and further details are available within the backmatter of the website version of this article.

1. Introduction

The system on a programmable chip (SOPC) architecture is superior to traditional devices. Compared with traditional microprocessors or field-programmable gate array (FPGA) devices, the SOPC model integrates the advantages of both. The microprocessor is realized using an embedded microcontroller unit (MCU), digital signal processor (DSP), and arm hardcore or softcore processor intellectual property (IP). In addition, with the rich IP resources and programmable logic resources in FPGA, it provides a good platform for software and hardware collaborative development technology. The SOPC system can comprehensively consider various situations of the whole system, reduce the connection delay between discrete devices under the same process and technical conditions, and significantly improve the reliability and performance of the system. For example, in Xilinx devices, the zynq-7045 is embedded with the advanced reduced instruction set computer machines (ARM) cortex-a9 hardcore microprocessor [1] and the xq7k325t chip integrated with a MicroBlaze softcore [2]. The m2s090ts-1fgg484m is embedded with an arm Cortex-M3 hardcore microprocessor in Microsemi devices [3]. These chips are programmable logic chips with embedded processor structures. The software and hardware interfaces are highly coupled, and the connection modes are diversified. For example, they are directly connected through a standardized bus protocol interface, or in a user-defined way, and may also contain some adhesive logic. At the same time, there are various ways to share data, including interrupt mode, memory mapping I/O mode, and special function register mode. According to different applications, the design scale and the complexity are also

different. In addition, because the programmable logic and embedded software in SOPC model design are usually described in different languages, and there are even a large number of black box IPs, the significant semantic gap between software and hardware makes verification more challenging. Traditional testing methods face serious challenges in SOPC model testing quality and efficiency.

The SOPC model combines the characteristics of software and logic, so it also integrates the verification methods of software and logic. The existing SOPC model dynamic test types include function test, performance test, interface test, boundary test, margin test, safety test, strength test, and other test types [4–6], which can cover the function requirements, performance requirements, safety requirements, and other requirements in the requirements specification from multiple dimensions, and can give specific data, such as data processing time, system response time, etc.

With the powerful function and wide application of SOPC architecture, the design scale of the system is becoming larger and more complex, and its security and reliability requirements are becoming more and more prominent. It is necessary to fully verify the SOPC model. Compared with ground software, once this SOPC software runs in orbit (the SOPC software runs on payload chips in space that will run in space for more than five years), it is difficult to locate the fault. Even if the fault can be located, a lot of costs would be required to upload and reconstruct it. Compared with traditional application-specific integrated circuit (ASIC) chips or FPGA chips, SOPC software usually includes one or more embedded processors, which can embed and run software programs and even operating systems on the chip.

The verification of the SOPC model consists of the evaluation of whether it meets the requirements from multiple dimensions such as functionality, performance, and security based on requirements. At present, the traditional SOPC model verification technology includes a simulation test, a static analysis, and a physical test. Static and simulation tests verify the software and logic, respectively, and cannot cover the area of software and hardware interaction. As shown in Figure 1, the logic is the yellow box, and the software (processing system) is the green box. If the software and hardware are tested together, physical testing needs to be used. Physical testing inputs excitation from the input end of the chip and observes the output from the output of the chip. The software and hardware coupling area cannot be observed and cannot achieve 100% coverage because it is inside the chip.

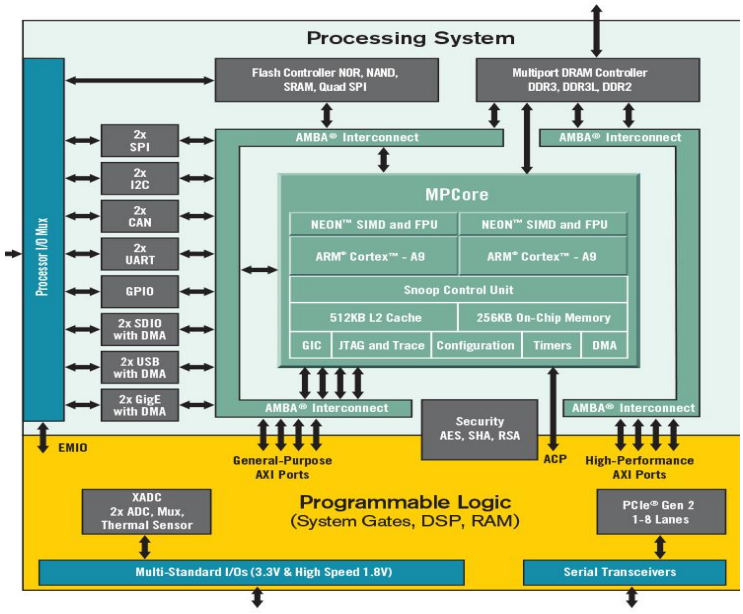

Figure 1. SOPC architecture.

The verification of the area of software and hardware interaction in physical tests is based on limited test scenarios, which can only prove the existence of errors and defects in some cases. However, for the test scenarios not involved in the test cases, the correctness of the system design cannot be guaranteed, so there may be some hidden defects that cannot be found. Therefore, dynamic verification is generally suitable for finding a large number of obvious design errors in the early stage of verification, but it is difficult to mine the hidden and subtle errors in complex design, such as system architecture errors, software, and hardware coordination errors, etc. The differences between traditional testing and formal testing are shown in Table 1. So when we focus on SOPC model verification, we use model-checking methods on the basis of traditional testing methods. This helps us find hidden defects in the code, like defects in software and hardware coupling.

Table 1. The differences between traditional testing and formal testing.

	Model Checking	**Traditional Test**
Test principle	Traverse all input paths	Limited testing case
Advantage	Comprehensive verification for a property	Provide information on whether the software meets the requirements based on the limited testing case principle
Disadvantage	Unable to obtain verification results when the code is complex	• Test results depend on test cases • Difficult to find the hidden and subtle errors in the complex design
SOPC test	• Verify the SOPC software and hardware interaction area • Verify the security properties of the SOPC software	• Discovering Problems in Dynamic Testing Based on Black Box • Analyzing possible defects in code based on static methods

The application practice of formal methods began in the 1970s and plays an extremely important role in the key field of safety. The most important feature of formal verification is that it is based on mathematical logic reasoning and proves whether the system design meets the system specification according to mathematical theory. For the existing test types and test methods of SOPC architecture, formal verification, as a powerful supplementary verification method, can make up for the shortcomings of the existing verification methods. By summarizing the key properties to be verified, it provides a good supplement to the existing dynamic test and static test, and effectively improves the quality of the software test and software reliability.

The research on model detection [7,8] began in the early 1980s when Clarke, Emerson, and others proposed CTL logic to describe the properties of concurrent systems, designed algorithms to detect whether a finite state system satisfies a given formula, and implemented a prototype system. The basic idea of model checking [9] is to express the temporal properties of a program or circuit using temporal logic formulas and to use finite state machines to represent the abstract structure of state transitions in a program or circuit [10,11]. The correctness of the temporal logic formulas is verified by traversing finite state machines. Model checking is an exhaustive search based on the system state space. The number of states often increases exponentially with the increase in concurrent components. Therefore, when a system has many concurrent components, it is not feasible to search its state space directly, which is the so-called state space explosion problem. The scale and complexity of aerospace SOPC software have doubled in the last ten years. At present, the scale of single-chip software has reached about 150,000 lines and involves the implementation of various complex protocols. For such a concurrent system as the SOPC model, the number of states often increases exponentially with the increase in concurrent components, so it is actually impossible to search its state space.

Aiming at the state space explosion of the SOPC model, our main contributions in this paper are listed as follows:

(1) A modeling method using SOPC architecture is proposed to solve the problem of multi-form IPs not being recognized by formal tools.
(2) A variable reduction method for the SOPC system is proposed, which can greatly reduce the number of states.
(3) A branch relation reduction method based on the verification property is proposed for the SOPC system. Through the reduction in branch relations, the state quantity and state transition relations of the model state space can be reduced.
(4) The experiment of property verification is carried out for the reduced SOPC model. The experiment result shows that the reduced model can greatly shorten the verification time.

The rest of this paper is organized as follows: The second section introduces the research achievements and existing problems related to state space compression. The third section analyzes the internal structure, characteristics, and scale of the SOPC model. The fourth section analyzes the connotation of state space explosion and the problem of state space explosion in the SOPC model. The fifth section proposes the method for establishing SOPC models and the mechanism for attribute verification. The sixth section proposes a code-level state space compression method. The seventh section validates the relevant attributes of the SOPC software and hardware interaction area based on the proposed method, and the feasibility of the method was demonstrated through the experimental results. The eighth section describes the conclusion.

2. Related Work

In [12], Billington et al. use distributed memory and computation for storing and exploring the state space of the model of a system. They present and compare different multi-thread, distributed, and cloud approaches to face the state space explosion problem. In [13], Partabian et al. propose an approach based on knowledge exploration for state space management in checking the reachability of complex software systems. The ensemble machine learning technique uses the Boosting method along with decision trees. This method generates k predictive models after sampling k times. Finally, it uses a voting mechanism to predict the labels of the final path. In [14], Kojima et al. propose a method based on symmetry reduction to reduce state space during model checking on secure routing protocols. They focus on the shapes of topologies. Loosely speaking, if the shape of the topology represented by the current state is the same as that of the searched state, the current state can be regarded as equivalent to the searched state by replacing nodes. In [15], in order to combat the state space explosion associated with BMC, Zhang et al. propose the method starts by combining module-level abstraction–refinement with slicing to reduce the size of the model under verification. In [16], Wu et al. propose a novel supervisor synthesis framework using automata learning and compositional model checking to generate the permissive local supervisors in a distributed manner. In [17], Wang et al. investigate the problems of applying the anti-chain approach to timed refinement checking and probabilistic refinement checking and show that the state space can be reduced considerably by employing the anti-chain approach. In [18], Shen et al. propose methods to accelerate hardware security verification and vulnerability detection through state space reduction. Specifically, we reduce the state space of the model by performing value reduction and transition relation reduction. The control flow and data-dependent graphs control the process of value reduction and transition relation reduction. In [19], Han et al. present an approach for schedulability analysis of Distributed Integrated Modular Avionics (DIMA) systems that consist of spatially distributed ARINC-653 multicore modules connected by a unified Avionics Full-Duplex Switched Ethernet (AFDX) network. In [20], to circumvent the state space explosion, Bortolussi et al. rely on stochastic approximation techniques, which replace the large model with a simpler one guaranteed to be probabilistically consistent. In [21], Konnov et al. introduced parametric interval counter abstraction that allowed us to verify the safety and liveness of threshold-based fault-tolerant distributed algorithms (FTDAs). In [22], Chai et al. focus on a major improvement in the analysis of reachability

properties in large-scale dynamical biological models. They introduce a hybrid approach, ASPReach, which combines static analysis and stochastic search to break the limits of existing approaches. In [23], Mikeev et al. propose a numerical integration algorithm to approximate the probability that a process conforms to a specification that belongs to a subclass of deterministic timed automata (DTA). They combat the state space explosion problem by using a dynamic state space that contains only the most relevant states. In [24], Alagar et al. present several techniques to tackle the state explosion problem at some consistent global state of a distributed system. In [25], in order to shrink the state space being observed during the model-checking process of TV software, a method is proposed that relies on using previous test logs to generate a partly non-deterministic user agent model. In [26], Zheng presents a new timing analysis algorithm for efficient state space exploration during the synthesis of timed circuits or the verification of timed systems. In [27], Xing et al. propose an analytical method based on sequential binary decision diagrams (SBDDs) for combinatorial reliability analysis of nonrepairable cold standby systems. Different from the simulation-based methods, the proposed approach can generate exact system reliability results.

In the above research methods, distributed computing and cloud computing indirectly improve the algorithm speed by utilizing the principle of parallel computing. The abstract method form is based on the original design and abstracted into a model with a smaller state space through various methods. The prerequisite for the application of these methods is that model verification tools can recognize the design. The SOPC model has many IP cores that cannot be recognized using formal tools, so relevant research cannot be applied.

3. Analysis of SOPC Software

SOPC software is the product of the development of FPGA to the system era, from the early programmable logic gate array to the on-chip system composed of interconnected IP through an on-chip high-speed bus. In SOPC software based on Advanced eXtensible Interface (AXI) bus protocol, data exchange is often realized between multiple master devices and multiple slave devices through a bus interconnection module. As shown in Figure 2, the master and slave devices can be IP core, programmable logic, and CPU softcore or hardcore. SOPC software has a large number of multi-form IP cores, such as netlist IP core, black-box IP core, encryption IP core, CPU IP soft core, CPU IP hardcore, and traditional register transfer level (RTL) IP core.

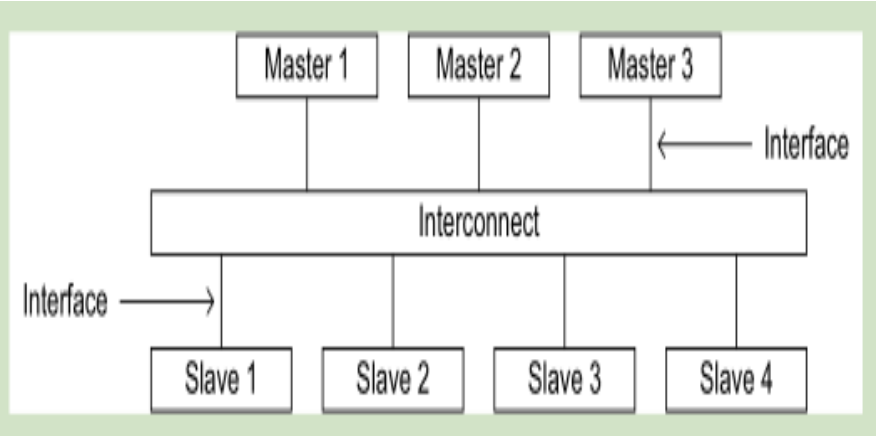

Figure 2. Multi-master and multi-slave interconnections.

When constructing software using SOPC architecture, a high-level language is generally used to describe the software function. After synthesis, place, and route, the code is converted into a logic circuit constructed from clock resources, place and route resources, registers, lookup tables, etc. For example, we use the Xilinx xq7k325t chip integrated with MicroBlaze softcore to realize the SOPC software, which includes network protocol,

FC-AE-1553 protocol, peripheral component interconnect express (PCIE) protocol, and other complex communication protocols, as well as high-speed and large-quantity data transmission and processing. FPGA's EDA tool (Vivado from Xilinx, San Jose, CA, USA, Libero from Microsemi, Irvine, CA, USA and so on) synthesizes RTL code [28] to a synchronous sequential software net. The resources occupied by the software are as follows (Figure 3):

Resource	Utilization	Available	Utilization %
LUT	112,099	203,800	55.00
LUTRAM	4619	64,000	7.22
FF	115,500	407,600	28.34
BRAM	156	445	35.06
DSP48	8	840	0.95
IO	131	500	26.20
GT	14	16	87.50
MMCM	2	10	20.00
PLL	1	10	10.00

Figure 3. The resources occupied by the FPGA software.

Advanced high-performance seven-series FPGA logic is based on real six-input lookup table technology. It includes a flip flop (FF), a lookup table(LUT), input–output (IO), and other components. From the synthesis result, we can see that the FPGA software has 115,500 flip flops and 112,099 $\times$ 6 logic lookup tables. In the worst case, a complete state diagram of the FPGA software is a complete graph with 2^{115500} States and $2^{115500} \times 2^{672594}$ transition relationships.

On the one hand, SOPC software faces multiform IPs that cannot be recognized using formal tools. On the other hand, SOPC software also faces the problem of state space explosion. Therefore, formal verification technology cannot be well applied to SOPC software at present.

4. Preliminaries

4.1. Temproal Logic

Temporal logic (temporal logic) identifies the temporal order in which events occur in a system and categorizes them according to a time model. Linear temporal logic (LTL) and computational logic (CTL) are commonly used languages for concurrent system specification and verification. Different temporal logics have their corresponding operators and their corresponding semantics. There are many branches of temporal logic, Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) are classical [29,30]. Temporal logic in the real-time design industry has evolved from early proprietary property languages such as Sugar from IBM [31] and Spec from Intel [32]. At the same time, there are a series of industry standards, including Accellera's Open Verification Library (OVL) [33], Property Specification Language (PSL) [34], and most recently the IEEE standard SystemVerilog Assertion (SVA) [35], which is the property language of a real-time system.

The IEEE standard temporal logic program SystemVerilog Assertion (SVA) is a combination of regular expressions and LTL formulas known as suffix implication formulas. A suffix implication formula is of the form $r \Rightarrow \phi$, where r is a regular expression and ϕ is either an LTL formula or another suffix implication formula. Intuitively, this formula states that whenever a prefix of a given computation path matches the regular expression r, the suffix of that path must satisfy ϕ. The formal definition of suffix implication is taken from [36]. Annex F in the SystemVerilog IEEE 1800-2012 language reference manual [34] gives the syntax and formal semantics of SVA, including the core operators for RE and LTL, rewriting rules for derivative forms and language sugaring, extensions including clocking,

and weak/strong semantics, etc. SVA formulas are built up from sequences. The core sequence syntax is:

$$\text{R::= b | R ##1 R | R ##0 R | R or R | R intersect R | R [}*0\text{] | R[}*1\text{:\$]}$$

where b is a Boolean expression, ## stands for the concatenation operation and intersects the conjunction operation.

4.2. Basic Definition

Definition 1. *A finite state machine: If M is a finite state machine, M = (S, I, O, f, g, s0). While S is a set of states, I is an input alphabet and O is an output alphabet. The transfer function and the output function are f, g. The initial state is s0.*

Definition 2. *Kripke structure: Atomic proposition (AP) is a set of finite atomic propositions. A Kripke structure K on AP is a 4-tuple k = (S, S_0, R, L), where S is a set of finite states; $S_0 \in s$ is the initial state set; $R \in S \times S$ represents all state transition relationships; and L: $S \to 2^{AP}$ indicates acceptable status.*

Definition 3. *Linear temporal logic (LTL) [36]: The LTL is a linear temporal logic formula. Formulas of LTL are built from a set of AP (atomic proposition) using the usual Boolean operators and temporal operators X ("next time") and $\cup$ ("until"). Given a set, AP, a formula is defined as follows:*

—true, false, p or $\neg$ p , for $p \in AP$

—$\psi \vee \varphi$, $\psi \wedge \varphi$,$X\psi$, $\psi \cup \varphi$, $\psi \tilde{\cup} \varphi$, where ψ and φ are an LTL formula.

We denote the length of a formula φ by $|\varphi|$. Given a Kripke structure M and an LTL formula φ, the model-checking aim is to determine whether all the computations of M satisfy φ. If this is the case, we denoted it by $M| = \varphi$, otherwise $M| \neq \varphi$.

Definition 4. *Deterministic Finite Automaton: The deterministic finite automaton will often be referred to by its acronym: deterministic finite automata (DFA). It shows that on each input there is one and only one state to which the automaton can transition from its current state. We often talk about a DFA in "five-tuple" notation: $M = (Q, \sum, \delta, q_0, F)$, where M is the name of the DFA, Q is its set of states, $\sum$ is its input symbols, δ is its transition function, q_0 is its start state, and F is its set of accepting states or final states, which is a subset of Q [37].*

Definition 5. *Nondeterministic Finite Automaton [37]: $M = (Q, \sum, \delta, q_0, F)$, where M is the name of the NFA, Q is its set of states, $\sum$ is its input symbols, δ is its transition function, q_0 is its start state, and F is its set of accepting states or final states, which is a subset of Q. The difference between the DFA and the NFA is in the type of "δ". For the NFA, "δ" is a function that takes a state and input symbol as arguments (like the DFA's transition function), but returns a set of zero, one, or more states (rather than returning exactly one state, as the DFA must). For example, a Büchi automaton is a nondeterministic finite automaton.*

Definition 6. *The "language" of the DFA [37]: The "language" of the DFA is the set of all strings that the DFA accepts. We can define the language of a DFA $A = (Q, \sum, \delta, q_0, F)$ as $L(A)$.*

In temporal logic model checking, we verify the correctness of a finite state system with respect to a desired behavior by checking whether a labeled state transition graph (system model) satisfies a temporal logic formula that specifies this behavior. Temporal logic model checking includes model checking based on automata theory and model checking based on CTL.

The principle of model checking based on automata theory is described in Section 5.2. Model checking based on CTL is to check whether the Kripke structure is a model of CTL for a given CTL formula. When applying model verification tools, the design is

generally described in VHDL and Verilog languages, while the property language is SVA. Model checking tools convert the design into a finite state machine first, then convert it to corresponding forms based on the principles of verification. For example, in model checking based on automata theory, the design and property are converted into design automata and property automata, respectively, for verification. In model checking based on CTL, the design is converted to a Kripke structure and the property is converted to CTL for verification.

4.3. State Space Explosion Problem

In the term state space, state refers to an ordered set of variables that can determine the minimum number of system states in a system. State space refers to the collection of all possible states of the system. The state space can simply be regarded as a space with a state variable as the coordinate axis, so the state of the system can be represented as a vector in this space. State space representation is a mathematical model that represents a physical system as a set of inputs, outputs, and states, and the relationship between inputs, outputs, and states can be described by many first-order differential equations.

The state variable of a system refers to the smallest subset of system variables that can represent the complete state of the system at any time. To represent the minimum value n of the required state variable for a system, it is usually also the order of the system's differential equation. If the system is represented by a transfer function, the minimum number of state variables is equal to the order of the denominator polynomial of the transfer function. The number of state variables in a circuit is often the number of states in the circuit.

The problem of state space explosion is that the number of states generated during model detection increases exponentially, far exceeding the memory storage and search capabilities of computers, which is one of the challenges faced by model detection as shown in Figure 4. The upper half of Figure 4 represents a typical SOPC structure, where the SOPC system connects multiple IPs through the AXI bus, and data exchange between IPs is achieved through bus interfaces. The lower half of Figure 4 represents the state transformation diagram of the SOPC model. The S represents a set of finite states, $S_0 \in S$ is the initial state set, and $R \subseteq S \times S$ represents all state transition relationships.

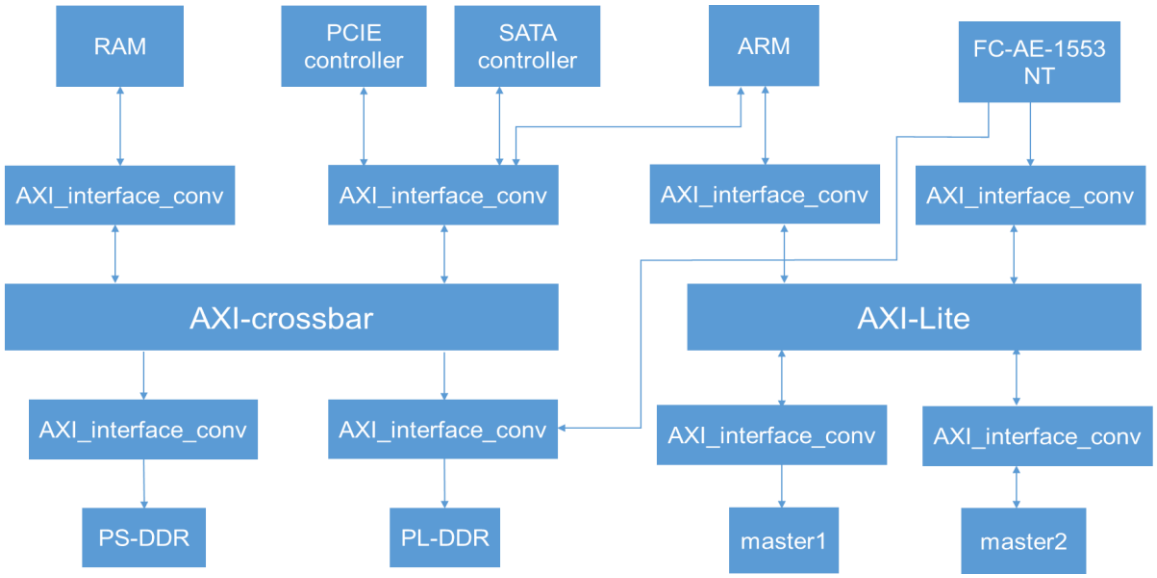

Figure 4. *Cont.*

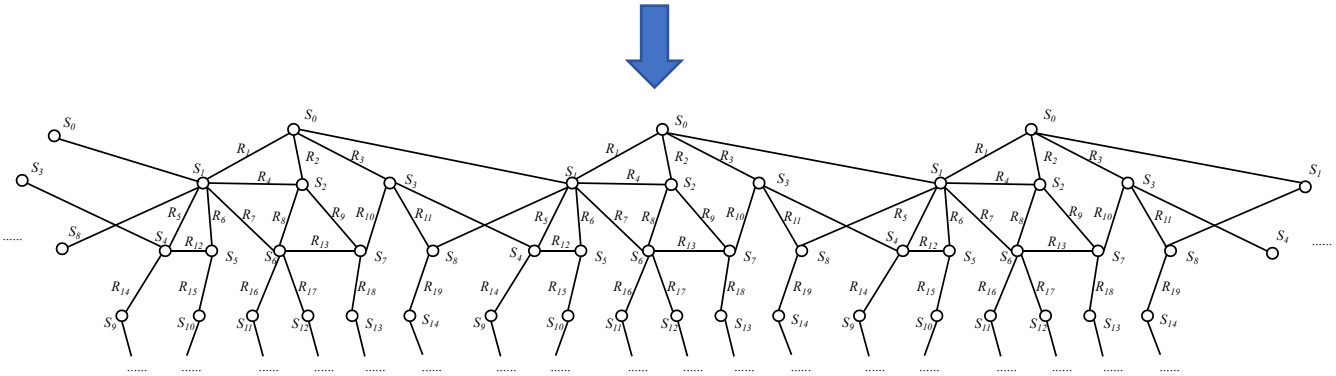

Figure 4. The problem of state space explosion.

From the analysis in the previous section, it can be seen that the scale of the SOPC model is getting larger and larger. Currently, model verification tools require more than 2 h which has exceeded the capacity of ordinary computing. At the same time, the SOPC model faces multiform IPs that cannot be recognized by formal tools. Therefore, formal verification technology cannot be well applied to the SOPC model at present.

Aiming at the problem of SOPC state space reduction, the method of this paper is to establish a system-level formal design SOPC model and propose a variable reduction method based on RTL code and a branch relationship reduction method based on verification properties. Through the above methods, the problem of SOPC state space explosion can be effectively tackled.

5. Modeling and Verification of the SOPC System

5.1. Modeling of SOPC System

This section will give the method of establishing a formal design model from the perspective of the SOPC system level as shown in Figure 5. An SOPC model may contain more than ten or more IPs according to the specific application design. Each IP core has specific functions. These IP cores are interconnected through the on-chip bus to achieve efficient data and information exchange.

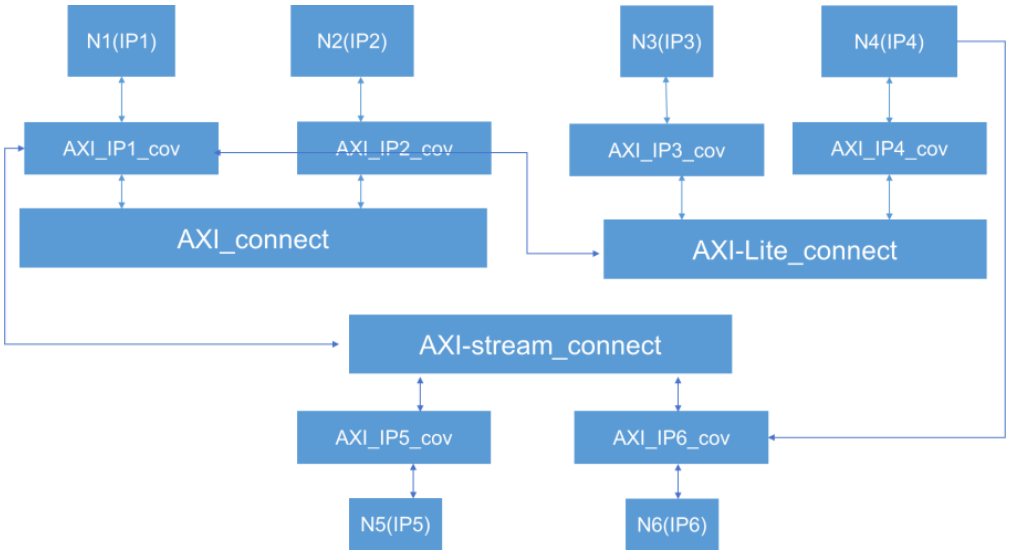

Figure 5. Block diagram of the SOPC system.

Suppose that a SOPC model design is expressed in M, and each independent IP in the SOPC model includes its own functions, interfaces, and interaction with other IPs, expressed as $m_1, m_2, \ldots, m_k \in M$. The on-chip bus interconnection module belonging to M

is represented as $M_{connect}$, and each IP is connected under the bus interconnection module with a standard interface unit. As shown in Formula (1):

$$M = M_{connect} \wedge m_1 \wedge m_2 \wedge \ldots \wedge m_k \tag{1}$$

Extract their behavior model n from different forms of IP, and the conversion from m to n should consider the design details required for verification, such as function items, internal and external data flow, interface behavior, interface timing relationship, etc. The conversion relationship from m to n can be expressed by Formula (2). After each IP is converted, an independent behavior model n is formed, and the behavior model N of the SOPC model is formed through the interaction between n and the bus. The conversion of the SOPC model is given by Formula (3).

$$m \xrightarrow{t} n \tag{2}$$

$$M \xrightarrow{t} N \tag{3}$$

Formula (4) shows that N is composed of $n_{connect}$ and multiple IPs. The formula shows that the behavior of the SOPC model is composed of many independent IP behavior models. The behavior level model of IP can be coded by using the combination of SystemVerilog Assertion (SVA) attributes, SystemVerilog (SV), and RTL codes. This paper considers that behavior modeling can set some configurable parameters, such as time parameters, instruction parameters, etc. Through the combination of these parameters, the behavior of each IP can be flexibly configured, which in turn can constrain the external behavior of the SOPC model.

$$N = n_{connect} \wedge n_1 \wedge n_2 \wedge \ldots \wedge n_k \tag{4}$$

$n_{connect}$ describes the specific behavior of the bus. If the bus behavior is modeled, it can be expressed as:

$$n_{connect} = (n, r_{data}, r_{addr}, w_{data}, w_{addr}, w_{ack}) \tag{5}$$

where n represents the IP on the bus, r_{addr} represents the read address transaction, r_{data} represents the read data transaction, w_{addr} represents the write address transaction, w_{data} represents the write data transaction, and w_{ack} represents the write response transaction.

The SOPC system-level abstract model is a complex network architecture. Various IPs are interconnected by different forms of AXI buses. The same IP can connect different AXI buses. Based on the SOPC system architecture in Figure 2 and Formula (5), we can express the SOPC system model as the following Formulas (6)~(9):

$$N_{AXI} = (IP1 \wedge AXI_{IP1_{cov}}) \wedge ((IP2 \wedge AXI_{IP2_{cov}}) \wedge AXI_{connect}) \tag{6}$$

$$N_{AXI_{Lite}} = (IP3 \wedge AXI_{IP3_{cov}}) \wedge ((IP4 \wedge AXI_{IP4_{cov}}) \wedge AXI_{lite_{connect}} \wedge (IP1 \wedge AXI_{IP1_{cov}})) \tag{7}$$

$$N_{AXI_{Stream}} = (IP5 \wedge AXI_{IP5_{cov}}) \wedge (IP6 \wedge AXI_{IP6_{cov}}) \wedge AXI_{stream_{connect}} \wedge (IP1 \wedge AXI_{IP1_{cov}}) \tag{8}$$

$$N = N_{AXI} \cup N_{AXI_{Lite}} \cup N_{AXI_{Stream}} \tag{9}$$

5.2. Property Verification

If both the implementation and specifications of the system are provided by the automaton, verifying whether the specifications of the system meet the implementation of the system requires actually checking whether the language contained in the implementation automaton belongs to a subset of the language contained in the property automaton. Property automaton provides all the behaviors allowed by the system, while implementation automaton provides the actual behavior of the system. If the language set implementing the automaton is included in the language of the property automaton, it means that any behavior implemented by the system is allowed by the specification. Otherwise, it means that the

behavior of implementing the automaton does not comply with the behavior defined by the property automaton. In the formal verification based on automaton theory, assuming that L(A) and L(P) represent the language accepted by the implementation of automaton A and property automaton P, respectively, testing $L(A) \subseteq L(P)$ is equivalent to testing whether $L(A) \cap \sim L(P)$ is empty. In this way, as shown in the following Equations (10)–(12), the combined automaton AP includes all the traces of implementation that do not meet the property. If the AP is a directed graph, detecting whether the AP has a trace that does not meet the specification is equivalent to detecting whether the AP contains at least one path from the initial state S0 to the accepted state as shown in Equation (13).

$$A = (Q_A, \sum_A, \delta_A, q_{0_A}, F_A) \tag{10}$$

$$P = (Q_p, \sum_p, \delta_p, q_{0_p}, F_p) \tag{11}$$

$$A \mid \mid P = C, C = (Q, \sum, \delta, q_0, F) \tag{12}$$

$$\begin{gathered} \text{While } Q = Q_A \times Q_p,\ \sum = \sum_A \cup \sum_p,\ q_0 = q_{0_A} \times q_{0_p},\ F = F_A \times F_p \\ \text{path} = q_0 \xrightarrow{\sigma 1} s_{\sigma 1} \xrightarrow{\sigma 2} s_{\sigma 2} \ldots\ldots \xrightarrow{\sigma f} s_{\sigma f}, s_{\sigma f} \in F \end{gathered} \tag{13}$$

As shown in Figures 6–8, we construct a finite directed graph AP whose nodes are an ordered pair. The first component of the ordered pair is a state of the implementing automaton A, and the second component is a subset of the states of the property automaton P. Because A is deterministic, it has only one initial state, and whenever a character is read, the implementing automaton A can only reach one next state. The automaton P is nondeterministic, which may transfer to multiple states after reading the character a. Additionally, s0 is the initial state of implement automaton A and S0 is a set of initial states. In concurrent systems, there may be multiple initial states. T0 is the initial state of the property automaton. If the automaton A can enter a state sj after accepting the input a, the automaton P can also accept the character a to enter one or more sub-state sets, tj. If tj and sj are both members of the state set, the “pass" is used to identify the state (sj, tj), otherwise the “fail” is used. If the automaton AP contains a path from the initial node to the failed node, it indicates that the language of automaton acceptance is outside the language of property automaton acceptance.

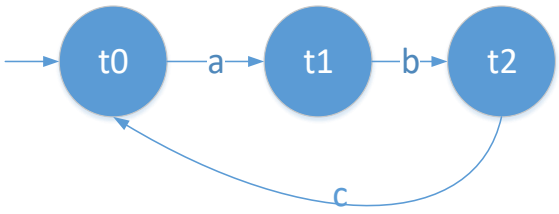

Figure 6. Property automaton P.

SVA is the abbreviation of SystemVerilog Assertions. From a linguistic perspective, it is a new temporal logic language derived from LTL and RE semantics. Its concise syntax can describe complex temporal behaviors, which traditional temporal logic languages do not have. Therefore, it is more suitable for describing the behavior of SOPC systems that are sensitive to timing. In any property model, a sequence is represented by a combination of multiple logical events, which can be a simple Boolean expression evaluated at the same clock edge or an event evaluated over several clock cycles. Many sequences can be sequentially combined to generate more complex sequences, and SVA provides a keyword “property” to represent these complex ordered behaviors.

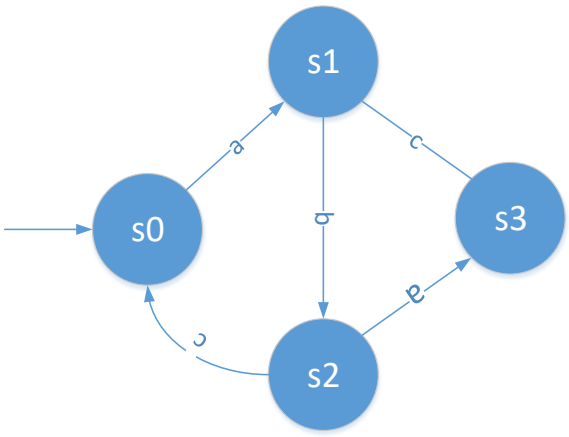

Figure 7. Implement automaton A.

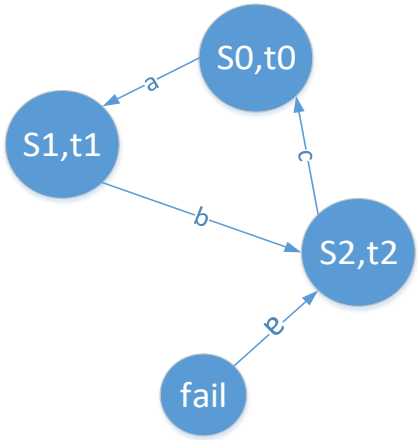

Figure 8. AP automaton.

We implemented model checking using the tool AveMC [38]. AveMC is an industrial-grade model checker; the designed inputs can be VHDL [39], Verilog [40], or SVA [41]. The tool automatically converts these inputs into the designed automata, and the input of the property is generally represented by SVA. The tool converts the property described by SVA into attribute automata and performs model verification according to the principles discussed at the beginning of this chapter.

6. Our Approach to State Space Reduction

6.1. State and State Transition Analysis

The state machine is composed of a state register and a combinational logic circuit. It can transfer the state according to the preset state based on the control signal. It is the control center that coordinates the action of relevant signals and completes specific operations. The state machine consists of a calculation model containing a set of states, a start state, a set of input symbols, a mapping input symbol, and a transition function from the current state to the next state.

In the actual digital system design, the finite state machine is usually used to establish the model of a sequential circuit, and in the formal verification, *Kripke* is generally used to represent the structure of the system, which is actually a deformation of the state transition diagram.

From Section 4.1, we can see that the state and state transition relationship of the system is the key factor in determining the complexity of state space. The state and state transition relationship of the digital circuit is equivalent or indirectly equivalent to the formal structure. Therefore, the state space explosion can start by reducing the state and state transition relationship of digital circuits.

In the digital circuit composed of the SOPC system, each bit of the register represents a status bit of the sequential circuit. Each register represents a set of states. For a sequential circuit with N bit registers and M bit inputs, the maximum number of valid states is 2^N, and there are 2^M input excitations for each state. In the worst case, a complete state diagram of a sequential circuit is a complete graph with 2^N States and $2^N \times 2^M$ transition relationships. The scale of complex synchronous sequential circuits is increasing day by day, and the state space is exponentially increasing.

6.2. Variable Reduction Based on RTL Code

In this section, we propose a variables reduction method based on RTL code, which can indirectly reduce the design state space. Variables in RTL code include the register, the wire, the input, and the output.

RTL codes are as follows (Table 2), where rec_nstate is a register, and wr_en_in, rstn, and wr_cmd_data_in are input.

Table 2. RTL codes.

RTL Codes

Reg[4:0] rec_nstate;
typedefenum logic [4:0] {SELFCHECK_TASK, RESET_TASK, DIR_TASK, SYN_TASK,
TIME_TASK, REC_TASK} state_enum;
always @(posedge clk or negedge rstn)
if(!rstn)
rec_nstate = SELFCHECK_TASK;
else if(wr_en_in && (wr_cmd_data_in[31:24] == 8'h01))
rec_nstate = SELFCHECK_TASK;
else if(wr_en_in && (wr_cmd_data_in[31:24] == 8'h02))
rec_nstate = RESET_TASK;
else if(wr_en_in && (wr_cmd_data_in[31:24] == 8'h04))
rec_nstate = DIR_TASK;
else if(wr_en_in && (wr_cmd_data_in[31:24] == 8'h08))
rec_nstate = SYN_TASK;
else if(wr_en_in && (wr_cmd_data_in[31:24] == 8'h10))
rec_nstate = TIME_TASK;
else
rec_nstate = rec_nstate;

The state (register) transition diagram based on RTL code is shown in Figure 9.

From the state transition in Figure 9, we can see that each assignment of the register is a separate state, and the transition between states is determined by conditional branching.

When we verify the properties of the SOPC system, we usually conduct a model check for each property. Since a single property does not involve all registers and variables of the design, if we only retain registers and variables related to the verification property, the state space of the design is reduced. This section proposes a method to reduce registers and variables from the perspective of the RTL code.

In state transition, if there is no path from the control variable to the state register, then this control path is empty. It is shown as P_c(var_ctrl, reg_state) = ϕ. As shown in Figure 9, P_c(wr_cmd_data_in, rec_nstate) $\neq \phi$.

In state transition, if there is no path from the assignment variable to the state register, then this assignment path is empty. It is shown as P_a(var_assign, reg_state) = ϕ. As shown in Figure 9, P_a(state_enum, rec_nstate) $\neq \phi$.

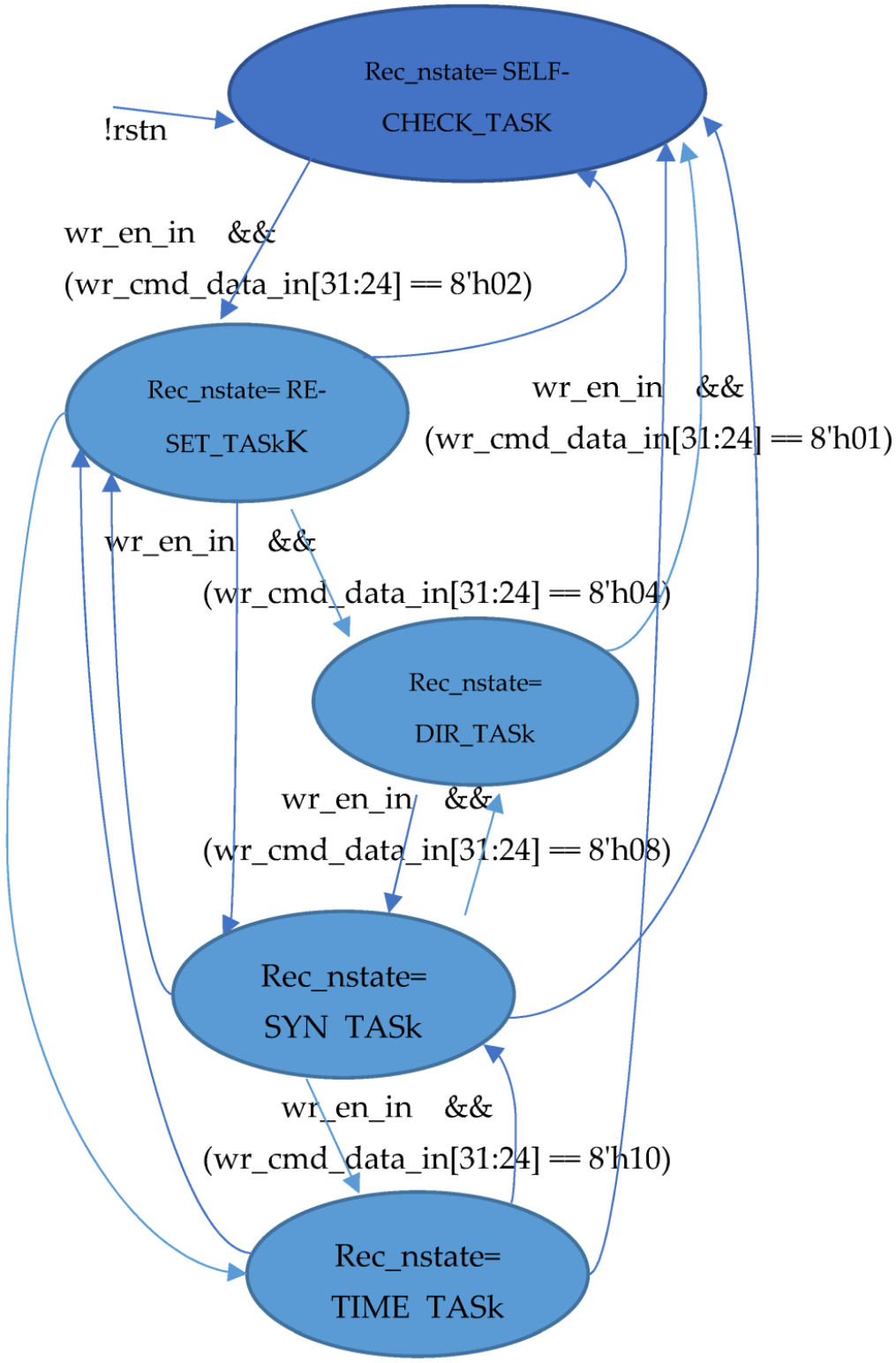

Figure 9. State transition diagram.

After analyzing the control path and assigning the path, we will discuss the reduction in design variables and registers by analyzing the variables of properties. An SVA property based on the RTL code in Figure 9 is shown below:

Suppose Rver is a verified register in a specification, then the verification variables for the above property include {rstn, rec_nstate, state_enmum}, where the rstn is the conditional variable, the state_enum is the assigning variable, and the rec_nstate is the state registers.

In this property, the variable rstn and state_enum in the design model can be retained, and other variables can be reduced; for example, the variable wr_en_in and the variable wr_cmd_data_in. In a real large-scale SOPC design, the complexity of the property and design would be much greater than in the example in Figure 9. The methods and steps of reduction are given below.

Definition 7. *Design variable minimum set: If R_d is the variable set of the design model, a minimum variable set $R_r \in R_d$ is found during the property verification, so that the states of the design model can be minimized, and the reduced design model should be equally equivalent to the design model before the reduction.*

The steps for variable reduction are shown in Figure 10:

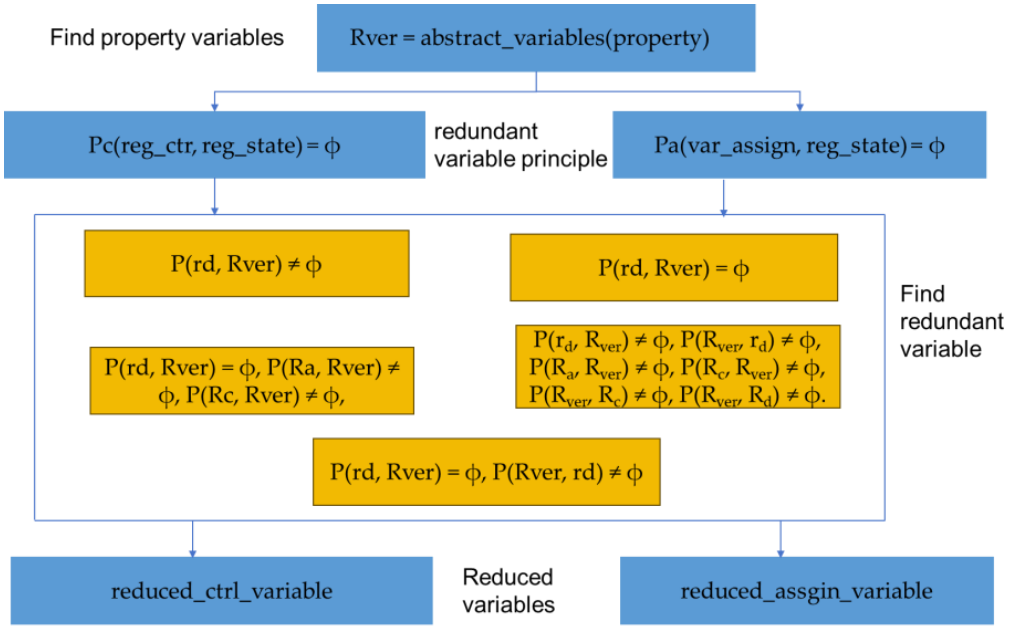

Figure 10. Variable reduction diagram.

R_{ver} is a set of verification variables that refer to the property. R_d is a set of model variables, $r_d \in R_d$. R_c is the control variable, and R_a is the assigning variable. These four types of variables have the following relationships.

As shown in Figure 11, $P(r_d, R_{ver}) \neq \phi$ indicates that variable r_d affects the property variables; therefore, r_d is an irreducible variable.

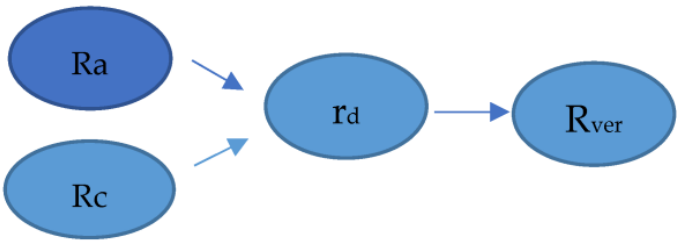

Figure 11. Variable interaction diagram.

As shown in Figure 12, $P(r_d, R_{ver}) = \phi$ indicates that the variable r_d does not affect the property variable; therefore, r_d is a reducible variable. Meanwhile, condition variables R_c and assignment variables R_a of the variable r_d are also reducible objects.

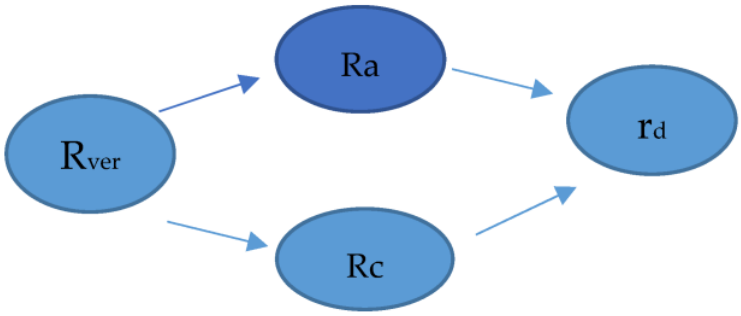

Figure 12. Variable interaction diagram.

As shown in Figure 13, $P(r_d, R_{ver}) = \phi$, $P(R_a, R_{ver}) \neq \phi$, and $P(R_c, R_{ver}) \neq \phi$ indicate that variable r_d does not affect the property variable; therefore, r_d is a reducible register. However, the condition variable R_c and the assignment variable R_a, which are simultaneously or one of the two affected property variables, are irreducible variables.

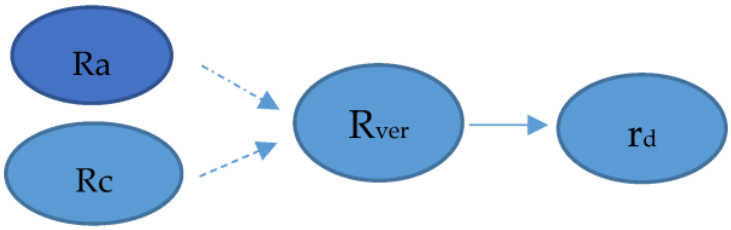

Figure 13. Variable interaction diagram.

As shown in Figure 14, $P(r_d, R_{ver}) \neq \phi$, $P(R_{ver}, r_d) \neq \phi$, $P(R_a, R_{ver}) \neq \phi$, $P(R_c, R_{ver}) \neq \phi$, $P(R_{ver}, R_c) \neq \phi$, and $P(R_{ver}, R_d) \neq \phi$ indicate that the variable r_d affects the property variable, and the property variable affects the variable r_d; therefore, r_d is an irreducible register. At the same time, the condition variable R_c and the assignment variable Ra are also irreducible variables.

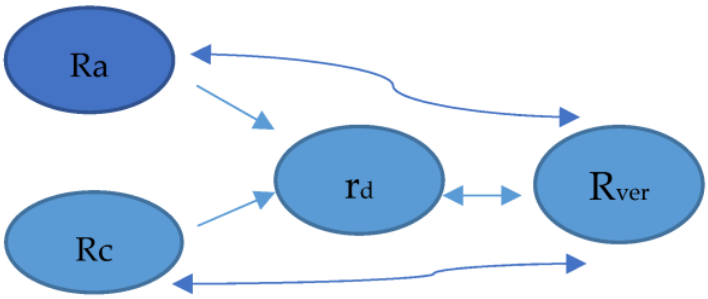

Figure 14. Variable interaction diagram.

As shown in Figure 15, $P(r_d, R_{ver}) = \phi$, and $P(R_{ver,} r_d) \neq \phi$ indicates that register r_d does not affect the property register; therefore, r_d is a reducible register. However, the condition variable R_c and the assignment variable R_a affect the property register; they are irreducible variables.

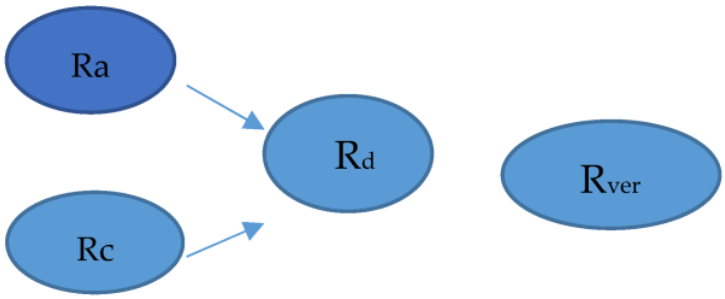

Figure 15. Variable interaction diagram.

6.3. Transform Relations Reduction Based on Property

In this section, we propose a state transition relation reduction method based on verification properties. Through the reduction in the transition relation, the model state space can be further reduced.

Through the analysis of the RTL code variables in the previous chapter, we reduced the variables of the model based on the analysis of verified property variables. After reduction, only the variables related to the verified property were left in the model. Based on the reduction in variables, we can further analyze the transform relations between variables. For the same number of variables, there can be many kinds of transform relations between variables. The idea of this section is to further reduce the state space of the model by reducing the transform relations irrelevant to the verification properties.

The method is shown in Figure 16. Firstly, we select the root node according to the verification property and then search the control branch and the assigned branch through the root node to form the complete network state diagram of the model.

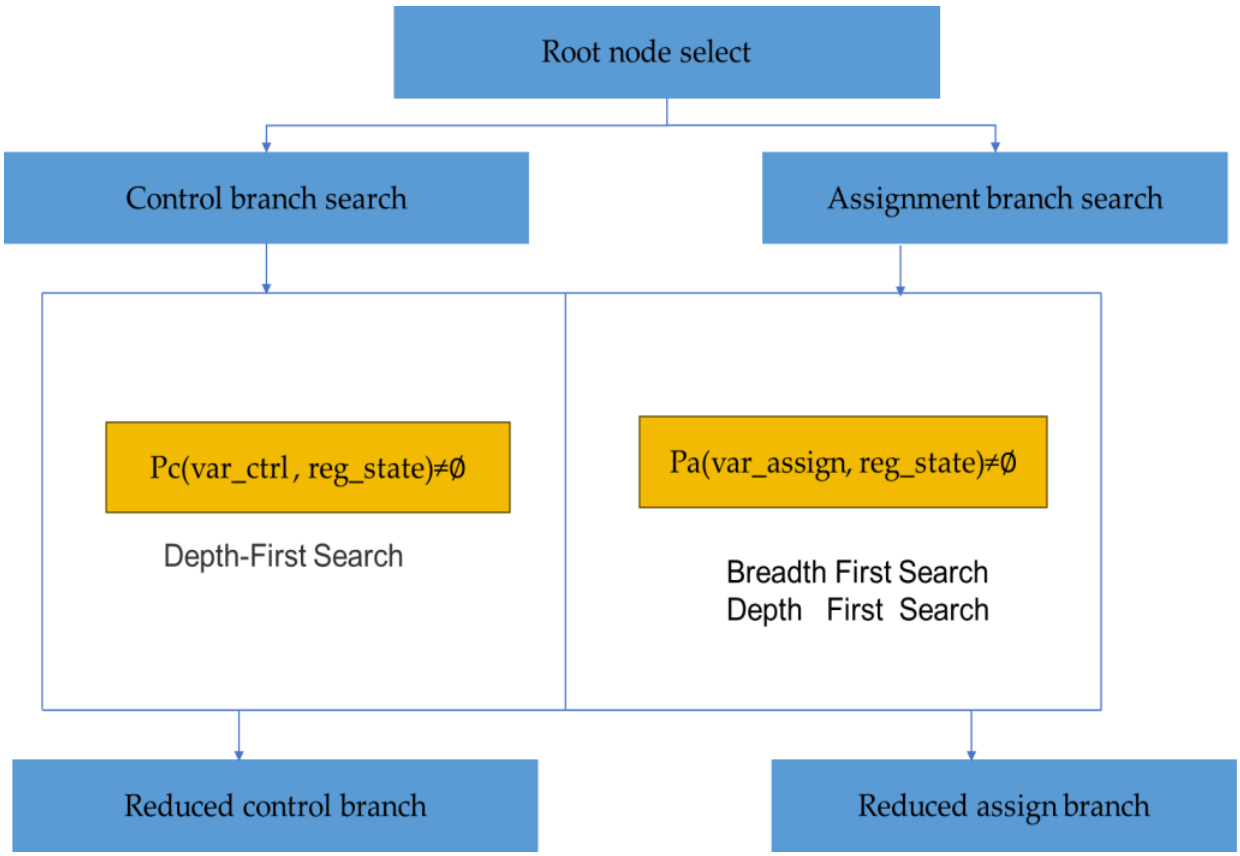

Figure 16. Transfer relations diagram.

1. Root node variable selection

The root node is based on the variables involved in the verification property. For example, the property is shown in Table 3.

Table 3. An SVA property based on the RTL code.

An SVA Property Based on the RTL Code
Property reset_check ($fell(rstn) | -> rec_nstate= SELFCHECK_TASK Endproperty

The root variables include the rstn and the rec_nastate. Because SELFCHECK_TASK is an enumeration constant, it is not included in the root node. The variable rstn is the input signal, so it is also not in the root node. Next, we use the root node as the starting point to find the control branch path and assignment branch path, respectively.

2 Control branch search

If v is the root node, we search the control path that controls the variable transforms. It is shown as P_c(var_ctrl, reg_state) $\neq \phi$. The var_ctrl includes the condition variables in the property. For example, if the condition variable of property reset_check is !rstn, then we start from the root node and find the condition branch !rstn, and then other branches are discarded.

When the branch variables involved in the property are complex, we start from the root node and use the Depth First Search (DFS) algorithm to search all the control paths related to the root node. When searching the first level variable, if the variable is an input signal or a constant, we stop searching the current node and return to other nodes at the same level. When the first level variable to be searched is a register, then we search the second level variable from the current node until the variable is an input or constant. As shown in Figure 17, we start from the root node and reverse search along one path to the end (root, variable 1). When we finish the first path, we search for the second path from the node register variable 2. For the two branches of variable 2, we start the search from the left branch until we find the input signal or constant node. After the search, we return to the node of variable 2 to start searching the right branch. We repeat this process until all control paths related to the root node are searched. The control path vector diagram of the root node is formed.

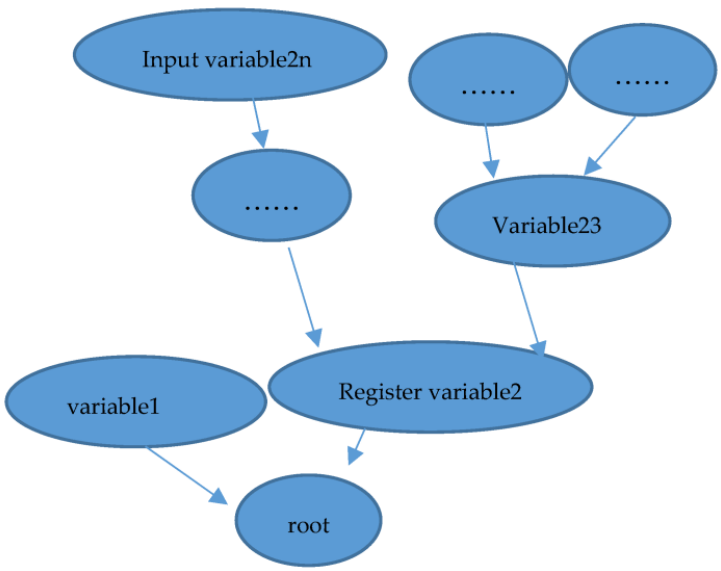

Figure 17. Control branch network.

3 Assignment branch search

An assignment transformation graph is a network graph for finding assignment variables related to verification variables. If v is the root node, we search the assignment path assigned to register v. It is shown as Pa(var_assign, reg_state) $\neq \phi$. The reg_state includes the register variables in the property. For example, the register variable of property reset_check is rec_nstate. If the property has conditions, for example, the property reset_check has a condition !rstn, then the assignment network graph only contains the assignment branches under property conditions. If there is no condition for the property, the assignment network diagram contains all assignment branches about the register variable. The RTL codes are shown in Table 4.

Table 4. The RTL codes.

The RTL Codes

//**********//
//task_num
//**********//
always @(posedge clk or negedge rstn) begin
if(!rstn) begin
task_num <= 8'd0; end
else if(inn_rst) begin
task_num <= 8'd0; end
else if((rec_cnt == 32'd1) && wr_en_in) begin
task_num <= wr_cmd_data_in[23:16]; end
else begin
task_num <= task_num; end
end

For example, if the property register variable contains the variable task_num, we search the task_num assignment network diagram. The task_num has three situations for assignment: one is constant 0, one is unchanged, and the other is wr_cmd_data_in[23:16], which is the input value. Therefore, when we search for the task_num assignment graph, there are only two layers: one layer is root nodes and the other is child nodes.

As shown in Figure 18, the actual assignment network structure diagram is much more complex than the above. In this network graph, there are many paths between register variables and assignment variables.

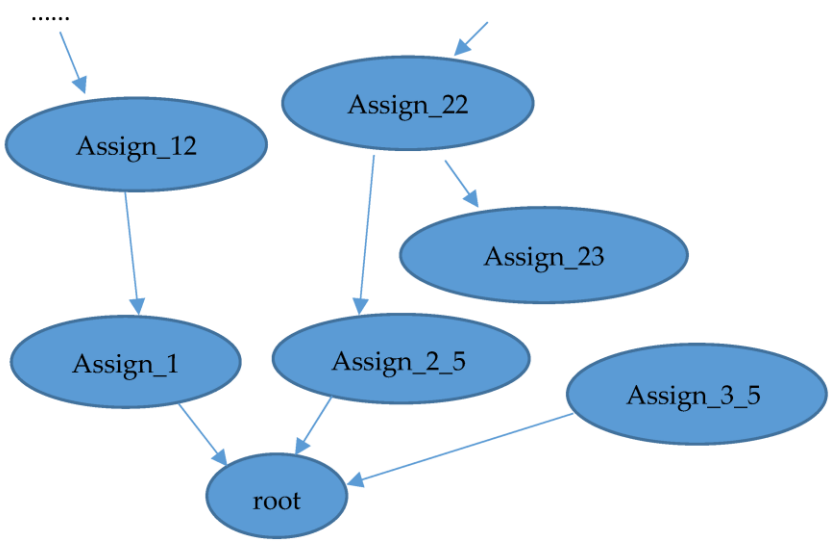

Figure 18. Assignment branch network.

When constructing the assignment network diagram, we use the combination of the Depth-First Search (DFS) algorithm and the Breath-First Search (BFS) algorithm. When traversing the variables at the first level, some variables have high weights, which can determine the value of the whole assignment statement. Then, we first traverse the variable using the depth-first search, and then traverse the variables at the same level in the breadth-first search, until all variables are traversed. The root node of all paths is the property register variable, and the endpoint of the child node is the input variable or constant of the module.

Through the construction of the control branch graph in Section 2 and the assignment branch graph in Section 3, we can build a complete design model for the property.

6.4. SOPC Reduced State Space Algorithm

This paper proposes a state space reduction algorithm for the SOPC software. The problem of state space explosion of the SOPC software can be solved through the construction of a system-level model in the SOPC system, the analysis of state and state transition, variable reduction, and transition relationship reduction. Relevant algorithms are shown in Algorithm 1. The algorithm is divided into the following three steps.

Algorithm 1: The formal verification framework for the SOPC software

```
INPUT: Design of an SOPC, A
OUTPUT: SOPC reduced model, D_fomal
/Step 1: SOPC design model construction*/
    M = Identify_need_model(A)
    FOR m ∈ M do
        If m ∈ CPU then
            m_formal = m = (s0, s, I, O, f, g);
        end
        else if m ∈ netlist then
            m_formal = m = (C, I, O, fc, g);
        end

        else if m ∈ black_box then
            m_formal = m = (I, O, g);
        end
            m_formal = m_formal ∪ M_no_need_model;

/Step 2: reduced variable */
    Rver = abstract_variables(property)
    reduced_ctrl_variable = Pc(reg_ctr, reg_state) = ϕ
    reduced_assign_variable = Pa(reg_assign, reg_state) = ϕ

    reduce_variable = reduced_ctrl_variable ∪ reduced_assgin_variable

/Step 3: reduce transfer relation */
    FOR {i = 1, i < n, i++} begin
        Net_ver_i_ctrl = DFS(ver_i)
        Net_ver_i_assgin = DFS(Ver_i) + BFS(Ver_i)
        Net_ver_i = Net_ver_i_ctrl ∪ Net_ver_i_assgin
    End
    D_fomal = convert_formal(Net_ver_1 ∪ Net_ver_2 ∪ . . . ∪ Net_ver_i)
```

The first step is the modeling of the SOPC system. When building the model, if there is an original RTL code, we use the RTL code as a formal design model, which includes VHDL and Verilog. If there is no RTL code, we can convert models from other languages, such as MATLAB and C, into VHDL or Verilog using conversion tools. If there are no models in other languages, we build behavior-level models based on requirements and

design documents, which can be used in VHDL, Verilog, or SVA. When building a design model manually, such as a CPU model, we do not need to describe the behavior of all CPUs. We only need to build the model for the behavior required, and we verify the consistency of the model by comparing the output under the same excitation after the construction is completed.

The purpose of modeling the SOPC system is to establish the minimum design model for formal verification. First, select all IPs of the SOPC system according to the verification properties, and discard the IPs irrelevant to the verification properties. Secondly, establish a model for the reserved IP. If it is an IP that cannot be recognized by the formal tool, the model of various IPs is given through the polymorphic IP abstract modeling technology. For example, for the softcore or hardcore of the CPU, it is necessary to establish the working mode and interaction behavior between the CPU and other IPs, especially the reading and writing of the public storage area, the reading and writing of the public register, and the interrupt processing. For netlist IP, this paper mainly establishes a model from the information flow between sub-modules and sub-modules. A sub-module is regarded as a node, and the interaction between nodes connects two modules. For black-box IP, this paper mainly establishes a behavior layer model from the input–output relationship, establishes a complete input set, and obtains the matching output by combining multiple conditions of the input set. For the IP that can be recognized by the formal tool, it is believed that the completed IP code can be retained through analysis. After the completion of the model of the multi-form IP, this paper can form a complete SOPC system-level model by building the model of the on-chip bus and the model of the standardized interface between the IP and the on-chip bus.

The second step is variable reduction. Firstly, we extract the variables from the property and then reduce the variables of the design model according to the six relationships among property variables, register variables, assignment variables, and control variables in the design model. After reduction, we find minimum variables set to meet the requirements of property verification.

In the third step, based on the variable reduction performed in the second step, we use the DFS algorithm to search the control branches of property register variables in the design model according to property conditions and construct the control network diagram. According to the property register and property conditions, the DFS and BFS algorithms are used to construct the assignment network diagram of the design model. After the construction of the control network and assignment network, we can obtain the design model to verify the property.

7. Experiment

In order to verify the state space reduction method proposed in our paper, we have conducted verification in the SOPC software. We compare the number of states, running time, and memory in the original design and the design after state reduction. The experimental results show that the algorithm proposed in this paper can be effectively applied to state space reduction in the SOPC software, and the algorithm does not require significant additional costs.

The algorithm proposed in Section 6.4 is verified in the experiment. Firstly, we applied the method in Section 5.1 to model the SOPC system and selected nine properties to verify the software and hardware interaction area of the SOPC model. Secondly, we applied the methods proposed in Sections 6.2 and 6.3 to compress space states. Finally, model checking was applied to the SOPC model, and we compared the results of the original SOPC model with the compressed SOPC model.

Our experiment was implemented using the AveMC model checking tool, which is an industrial-grade model checker. The description language of the design is SVA, Verilog, Very-High-Speed integrated circuit hardware description language (VHDL), or SystemVerilog, and the property language is Verilog and SVA.

7.1. Construction of SOPC System-Level Original Model

As shown in Figure 19, SOPC uses the AXI bus as the on-chip bus to connect two hosts and five slaves. One host is ARM, and the other host is the network terminal (NT) of the FC-AE-1553 bus. One slave is dual-port random access memory (RAM), and the other slave is a programmable logic double data rate synchronous dynamic random access memory (PL-DDR), a processing system-DDR (PS-DDR), and two serial advanced technology attachment (SATA) controllers. There are six types of IPs interconnected with the bus. According to the form of the interconnected IPs, this paper divides them into three types. The first type is CPU hardcore: ARM core. The second category is RTL code IP: RAM core, SATA_ Contrller core, PS_ DDR core, PL_ DDR core. The third type is encryption IP: FC-AE-1553 NT core. We can abstract and formalize the first type of IP core and the third type of IP core.

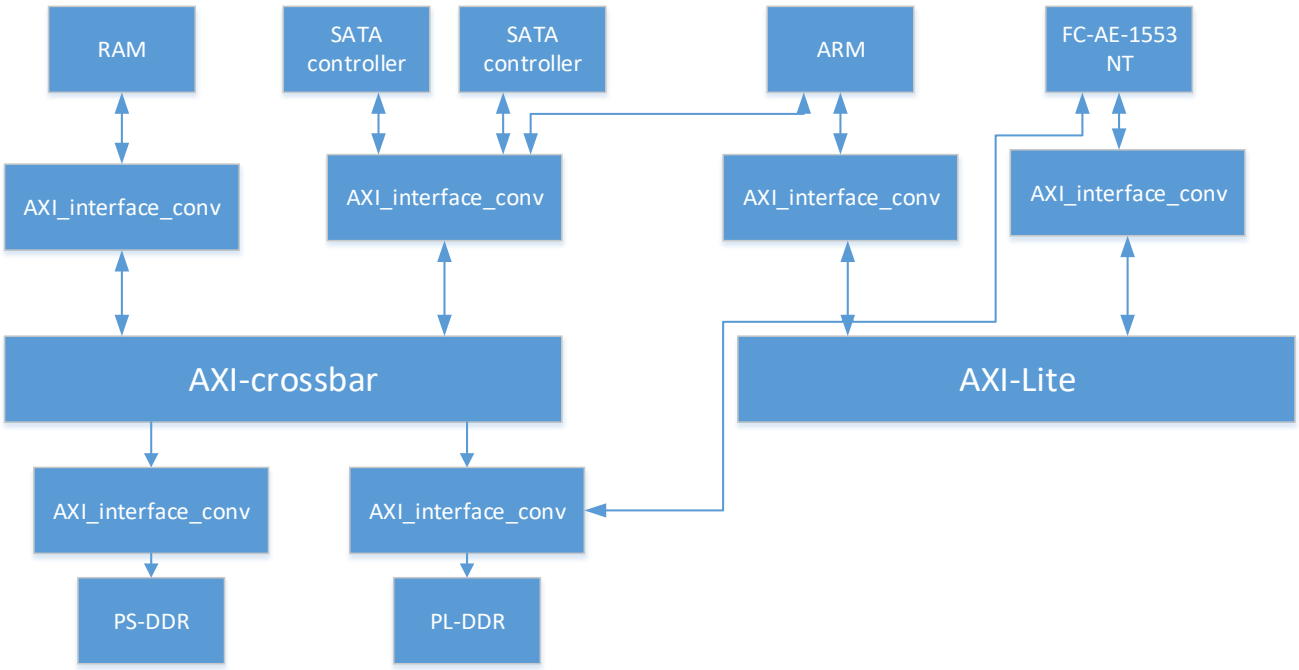

Figure 19. Original SOPC system.

The bus selected by the experimental object is AXI4 and AXI4-lite. At the bus level, AXI4-lite and AXI have their own RTL-level IP cores, so their respective RTL-level IP cores are directly used as the initial formal design model. At the interface level, the first and third types of IP without RTL-level source code are summarized. This paper adds the RTL-level code that summarizes the interface conversion as the behavior of the CPU and FC-AE-1553 NT model interacting with the bus, as shown below (Figure 20):

Based on the above model construction, this paper defines the model of the initial SOPC system as the input of the state space reduction algorithm and generates the reduced model of the SOPC system.

7.2. Formal Property Select

SOPC hardware and software cooperation refers to the cooperation between logic and software to complete transactions. The SOPC system cannot perform hardware and software co-simulation, so it is impossible to perform white box simulation analysis of the hardware and software interaction area. The board-level test is a limited scenario test, and the external input excitation cannot ensure effective coverage of the software and hardware interaction area, so the SOPC software and hardware interaction area is the risk point of verification, which brings hidden problems. This paper focuses on the verification of the SOPC software and hardware interaction part, so the design properties focus on the

security of the software and hardware interaction part. According to the characteristics of the experimental object, this paper selects the nine verification properties shown in Table 5.

```
// I/O Connections assignments

assign S_AXI_AWREADY          = axi_awready;
assign S_AXI_WREADY           = axi_wready;
assign S_AXI_BRESP  = axi_bresp;
assign S_AXI_BVALID = axi_bvalid;
assign S_AXI_ARREADY          = axi_arready;
assign S_AXI_RDATA  = axi_rdata;
assign S_AXI_RRESP  = axi_rresp;
assign S_AXI_RVALID = axi_rvalid;
// Implement axi_awready generation
// axi_awready is asserted for one S_AXI_ACLK clock cycle when both
// S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_awready is
// de-asserted when reset is low.

always @( posedge S_AXI_ACLK )
begin
 if ( S_AXI_ARESETN == 1'b0 )
   begin
     axi_awready <= 1'b0;
     aw_en <= 1'b1;
   end
 else
   begin
     if (~axi_awready && S_AXI_AWVALID && S_AXI_WVALID && aw_en)
       begin
         // slave is ready to accept write address when
         // there is a valid write address and write data
         // on the write address and data bus. This design
         // expects no outstanding transactions.
```

Figure 20. Interface software RTL code.

Table 5. Formal properties.

Type	Property
Software and hardware interaction area	(1) Does not read and write the same address of dual-port RAM at the same time; (2) Two masters cannot write to the same address at the same time; (3) Does not read and write the same register at the same time; (4) The software reads the value of the updated logic register within N ms; (5) The interrupt status register does not lose an interrupt; (6) The time interval of the interrupt request shall not be less than 1 s; (7) The last interrupt ID is cleared before the next interrupt; (8) One bit of interrupt register cannot be read and written at the same time; (9) The interrupt status register does not lose an interrupt.

We can use SVA language to describe the properties to be verified as shown in Table 6.

Table 6. Formal properties description.

Type	Property Description	Property
Read_Write conflict	Does not read and write the same address of dual-port RAM at the same time.	!(Read&&write&&(Read_addr==Write_addr))
	Two masters cannot write to the same address of shared storage at the same time.	!(master1_wr&&master2_wr&&(master1_wraddr==master2_wraddr))
	Does not read and write the same register at the same time.	!(Read&&write &&(rd_regaddr==wr_regaddr))
	The software reads the value of the updated logic register within N ms.	$changed(reg) |=> #[0:100] reg_read
Read_write interrupt conflict	The interrupt status register does not lose an interrupt.	//property Inter1_in&&no_other_inters |=>inter_status[1];

	The time interval of the interrupt request shall not be less than 1 s.	Always @(posedge clk or posedge rst)
		If(!rst) counter<=0;
		Else if(int_req||counter ==TIME) counter<=0;
		Else counter <=counter +1;
		Assign flag = (count !=0)
		//property
		(int_req & flag)

	The last interrupt ID is cleared before the next interrupt.	Always @(posedge clk or posedge rst)
		If(!rst) set_flag <=0;
		Else if(interruptid_clear) set_flag <=0;
		Else if(interruptid_set) set_flag <=1;
		//property
		(!set_flag)&&interrupt

	One bit of interrupt register cannot be read and written at the same time.	assign rd_intr0=rd &intr_reg[0];
		assign wr_intr0=wr &intr_reg[0];
		//property
		!(rd_intr0&& wr_intr0)
		assign rd_intr1=rd &intr_re [1];
		assign wr_intr1=wr &intr_reg[1];
		//property
		!(rd_intr1&& wr_intr1)

	The interrupt status register does not lose an interrupt.	Inter1_in&&noother_inters |=>inter_status[1];

7.3. Model Checking Result Analysis

This original design was implemented using Xilinx Kintex-7 series FPGA chips, the language was Verilog, and the number of lines of code was about 70,000 lines. After the front-end code was synthesized, placed, and routed, a synchronous sequential software composed of logic resources, wiring resources, clock resources, storage resources, IP resources, etc., was obtained. The resources occupied by this design are shown in Figure 21.

Resource	Utilization	Available	Utilization %
FF	221,340	508,400	43.54
LUT	182,304	254,200	71.72
Memory LUT	5476	90,600	6.04
I/O	144	500	28.80
BRAM	393.5	795	49.50
DSP48	386	1540	25.06
BUFG	31	32	96.88
MMCM	3	10	30.00
PLL	1	10	10.00
GT	16	20	80.00

Figure 21. The resources occupied by the original design.

The resource list in the figure above includes flip flop (FF), lookup table (LUT), input–output (IO), and other components. From the synthesis result, we can see that the FPGA software has 221,340 flip flops and 182,304 × 6 logic lookup tables. Based on the third chapter theory analysis of the number of states, if this SOPC software does not perform state reduction, 2^{221340} states and $2^{221340} \times 2^{71}$ transition relationships need to be traversed in formal verification.

After the model of the CPU was established, we integrated the CPU model and the SOPC logic code to form a complete formal SOPC design model. In the comparative experiment, based on the original SOPC formal design model and the formal design model after variables reduction and transfer relations reduction, we conducted model checking on the five properties in the previous section, and the test results are shown in Tables 7 and 8.

Table 7. Running Results of the experiment.

Property	Pass *	Original Formal Design Model		Reduction Formal Design Model	
		Memory (MB)	*Running Time* (s)	*Memory* (MB)	*Running Time* (s)
P1	√	NA	NA	1056	2954
P2	×	NA	NA	1024	2410
P3	√	NA	NA	1114	3202
P4	√	274	810	124	44
P5	√	214	702	122	40
P6	×	680	3450	685	211
P7	√	1036	3228	780	130
P8	×	NA	NA	1024	3581
P9	√	362	1012	178	68

* "×" means property verification failed; "√" means property verification passed; and NA means no detection.

Table 8. State Reduction Results of the experiment.

Code Line	Property	Original States	Reduced Register (States)	Ratio
77,892	P1	221,340	90,882	41%
	P2		77,468	35%
	P3		97,792	44%
	P4		11,079	5%
	P5		10,495	4.74%
	P6		34,773	15.7%
	P7		29,758	13.44%
	P8		79,984	36%
	P9		17,788	8%

In the experiment process, we first performed the variable reduction method on the original model. After the reduction, we performed transfer relations reduction.

We compared the states of the original design with the states of the reduced design and compared the run time with the memory of different properties.

In Table 7, the first two columns, "property" and "pass", show verified properties and the result of property verification. The third to sixth columns, respectively, show the total running time needed to verify each property and the peak memory. In Table 7, the results of the original formal design model test are shown; property 1 and property 2 have no results. In the experiment we chose 2 h as the time limit; beyond 2 h, the property cannot be verified.

From the memory usage and runtime of property verification, we can see that the runtime is significantly reduced. Moreover, in experiments with property 1 and property 2, we set their timeout as two hours. While the properties cannot be verified in the original design, they can be verified after state reduction.

For different properties, if the design state space involved in property A is larger than the design state space involved in property B, then the time to detect property A is longer

than the time to detect property B. For example, in Table 7, since property 4 involves more design states than property 3, the detection time is also longer than that of property 3.

In Table 8, the first two columns, "code line" and "property", show the original design lines and verified properties. The third column gives the states of the original design, the fourth column gives reduced states, and the fifth column gives the ratio of state reduction. Because the two state reduction algorithms proposed in this paper are based on properties, the number of design states based on different properties is also different.

The model reduction method proposed in this paper is a lossless abstract method, which is mainly based on a search of design variables related to properties and a search of transfer relationships to reconstruct the model of the design. Therefore, it is more comprehensive in accuracy than general abstract models at the signal level.

8. Conclusions

Aiming at the formal verification problem encountered in the SOPC software widely used in the high-reliability field, this paper proposes a modeling method of the SOPC system and a state space reduction method for SOPC software. We provide a variable reduction method for the SOPC system and a branch relation reduction method based on verification properties. Through the reduction in branch relations, the state quantity and state transition relations of the model state space can be reduced. The proposed methods are evaluated in the actual project. The experimental results have demonstrated that our proposed methods can significantly reduce the complexity of the model and thus the formal verification time for SOPC software. The method proposed in this paper can improve the coverage of SOPC testing in areas where traditional testing coverage is insufficient; it is a supplement to testing.

When reducing state space, this paper focuses on VHDL and Verilog, which can be synthesized. In fact, high-level languages such as SystemVerilog and SystemC can also be used when constructing models. For this high-level language, although similar methods can be used for variable and state transition relationship reduction at the code level, there are still more details to consider at the practical reduction level. For example, the transformation from high-level language to state transition graph, the definition and reduction scope of high-level language variables, and the reduction method of high-level language for loops, etc. We hope that further research can be applied to a wider range of model language expressions.

Author Contributions: Study Conception and Design: J.W. and S.Z.; Model: S.Z., P.X. and X.W.; Methodology: S.Z., J.W., S.Z. and L.K.; Analysis and Interpretation of Results: S.Z., J.W., P.X., X.W. and L.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The data presented in this study are available on request from the corresponding author.

Acknowledgments: The authors would like to thank the anonymous reviewers for reviewing and providing insightful suggestions in the writing of this paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Zynq 7000 SoC. Available online: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html (accessed on 3 November 2023).
2. Kintex 7 FPGA Family. Available online: https://www.xilinx.com/products/silicon-devices/fpga/kintex-7.html (accessed on 3 November 2023).
3. M2S090TS-1FGG484M | Microsemi. Available online: https://www.microsemi.com/existing-parts/parts/143678#overview (accessed on 3 November 2023).
4. Xiong, W.; Shi, W.; Dong, J.; Bai, Z.; Tian, D. Design of embedded automatic test system for radar transmitter. In Proceedings of the IEEE 2011 10th International Conference on Electronic Measurement & Instruments, Chengdu, China, 16–19 August 2011.

5. Chen, S.; Zhou, Y.; Zhu, D.; Guo, S. Design of high-speed Boundary-scan master controller base on SOPC. In Proceedings of the 2011 Second International Conference on Mechanic Automation and Control Engineering, Inner Mongolia, China, 15–17 July 2011.
6. Huang, Q.; Chang, S.; Peng, J.; Mao, X.; Zhou, Y.; Wang, H. Design of high-speed Boundary-scan master controller base on SOPC. *IEEE Trans. Instrum. Meas.* **2012**, *61*, 2469–2475. [CrossRef]
7. Clarke, E.M.; Emerson, E.A.; Sistla, A.P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst. (TOPLAS)* **1986**, *8*, 244–263. [CrossRef]
8. Emerson, E.A.; Halpern, J.Y. Decision procedures and expressiveness in the temporal logic of branching time. In Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing, San Francisco, CA, USA, 5–7 May 1982.
9. Huth, M.; Ryan, M. *Logic in Computer Science: Modelling and Reasoning About Systems*; Cambridge University: Cambridge, UK, 2004.
10. Phyo, Y.; Do, C.M.; Ogata, K. A support tool for the L+ 1-layer divide & conquer approach to leads-to model checking. In Proceedings of the 2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC), Madrid, Spain, 12–16 July 2021.
11. Weyns, D.; Iftikhar, U.M. ActivFORMS: A formally founded model-based approach to engineer self-adaptive systems. *ACM Trans. Softw. Eng. Methodol.* **2023**, *32*, 1–48. [CrossRef]
12. Billington, J.; Gallasch, G.E.; Kristensen, L.M.; Mailund, T. Exploiting equivalence reduction and the sweep-line method for detecting terminal states. *IEEE Trans. Syst. Man Cybern.-Part A Syst. Hum.* **2004**, *34*, 23–37. [CrossRef]
13. Partabian, J.; Rafe, V.; Parvin, H.; Nejatian, S. An approach based on knowledge exploration for state space management in checking reachability of complex software systems. *Soft Comput.* **2020**, *24*, 7181–7196. [CrossRef]
14. Kojima, H.; Yanai, N. A model checking method for secure routing protocols by SPIN with state space reduction. In Proceedings of the 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), New Orleans, LA, USA, 18 May 2020.
15. Zhang, Y.; Chakrabarty, K.; Peng, Z.; Rezine, A.; Li, H.; Eles, P.; Jiang, J. Software-based self-testing using bounded model checking for out-of-order superscalar processors. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2020**, *39*, 714–730. [CrossRef]
16. Wu, B.; Zhang, X.; Lin, H. Permissive supervisor synthesis for Markov decision processes through learning. *IEEE Trans. Autom. Control* **2019**, *64*, 3332–3342. [CrossRef]
17. Wang, T.; Chen, T.; Liu, Y.; Wang, Y. Anti-chain based algorithms for timed/probabilistic refinement checking. *Sci. China Inf. Sci.* **2018**, *61*, 052105. [CrossRef]
18. Shen, L.; Mu, D.; Cao, G.; Qin, M.; Zhu, J.; Hu, W. Accelerating hardware security verification and vulnerability detection through state space reduction. *Comput. Secur.* **2021**, *103*, 102167. [CrossRef]
19. Han, P.; Zhai, Z.; Nielsen, B.; Nyman, U.; Kristjansen, M. Schedulability analysis of distributed multicore avionics systems with uppaal. *J. Aerosp. Inf. Syst.* **2019**, *16*, 473–499. [CrossRef]
20. Bortolussi, L.; Lanciani, R. Schedulability Analysis of Distributed Multicore Avionics Model checking Markov population models by stochastic approximations. *Inf. Comput.* **2018**, *262*, 189–220. [CrossRef]
21. Konnov, I.; Veith, H.; Widder, J. On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability. *Inf. Comput.* **2017**, *252*, 95–109. [CrossRef]
22. Chai, X.; Ribeiro, T.; Magnin, M.; Roux, O.; Inoue, K. Static analysis and stochastic search for reachability problem. *Electron. Notes Theor. Comput. Sci.* **2020**, *350*, 139–158. [CrossRef]
23. Mikeev, L.; Neuhäußer, M.R.; Spieler, D.; Wolf, V. On-the-fly verification and optimization of DTA-properties for large Markov chains. *Form. Methods Syst. Des.* **2013**, *43*, 313–337. [CrossRef]
24. Alagar, S.; Venkatesan, S. Techniques to tackle state explosion in global predicate detection. *IEEE Trans. Softw. Eng.* **2001**, *27*, 704–714. [CrossRef]
25. Comert, F.; Ovatman, T. Attacking state space explosion problem in model checking embedded TV software. *IEEE Trans. Consum. Electron.* **2015**, *61*, 572–579. [CrossRef]
26. Zheng, H. Compositional reachability analysis for efficient modular verification of asynchronous designs. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2010**, *29*, 329–340. [CrossRef]
27. Xing, L.; Tannous, O.; Dugan, J.B. Reliability analysis of nonrepairable cold-standby systems using sequential binary decision diagrams. *IEEE Trans. Syst.* **2011**, *42*, 715–726. [CrossRef]
28. Sozzo, E.D.; Conficconi, D.; Zeni, A.; Salaris, M.; Sciuto, D.; Santambrogio, M.D. Pushing the level of abstraction of digital system design: A survey on how to program FPGAs. *ACM Comput. Surv.* **2022**, *55*, 1–48. [CrossRef]
29. Hu, J.; Chen, S.; Chen, D.; Kang, J.; Wang, H. Model-based Safety Analysis for an Aviation Software Specification. *Int. J. Perform. Eng.* **2020**, *16*, 238–254.
30. Langenfeld, V.; Dietsch, D.; Westphal, B.; Hoenicke, J. Scalable Analysis of Real-Time Requirements. In Proceedings of the 2019 IEEE 27th International Requirements Engineering Conference (RE), Jeju Island, Republic of Korea, 23–27 September 2019.
31. Beer, I.; Ben-David, S.; Eisner, C.; Fisman, D.; Gringauze, A.; Rodeh, Y. The Temporal Logic Sugar. In Proceedings of the Computer Aided Verification: 13th International Conference (CAV), Paris, France, 18–22 July 2001.

32. Armoni, R.; Fix, L.; Flaisher, A.; Gerth, R.; Ginsburg, B.; Kanza, T.; Landver, A.; Mador-Haim, S.; Singerman, E.; Tiemeyer, A.; et al. The forspec temporal logic: A new temporal property-specific logic. In Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Grenoble, France, 8–12 April 2002.
33. OVL (Open Verification Language). Available online: https://www.eda.org/downloads/standards/ovl (accessed on 16 November 2023).
34. PSL, Standard for Property Specification Language (PSL). Available online: https://www.eda.org/downloads/ieee (accessed on 16 November 2023).
35. IEEE Standard for Systemverilog—Unified Hardware Design, Specification, and Verification Language. Available online: https://accellera.org/downloads/ieee (accessed on 16 November 2023).
36. Ben-David, S.; Copty, F.; Fisman, D.; Ruah, S. Vacuity in practice: Temporal antecedent failure. *Form. Methods Syst. Des.* **2015**, *46*, 81–104. [CrossRef]
37. Hopcroft, J.E.; Motwani, R.; Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation*, 3rd ed.; Addison-Wesley Publishing Company: Boston, MA, USA, 2006.
38. AveMC, a Formal Verification Platform. Available online: https://www.arcas-da.com/EN/html/products/AveMC.html (accessed on 26 November 2023).
39. Ashenden, P.J. *The Designer's Guide to VHDL*; Margan Kaufmann: San Francisco, CA, USA, 2002.
40. Thomas, D.; Moorby, P. *The Verilog® Hardware Description Language*; Springer Science & Business Media: Berlin, Germany, 2008.
41. Vijayaraghavan, S.; Ramanathan, M. *A Practical Guide for SystemVerilog Assertions*; Springer Science & Business Media: Berlin, Germany, 2005.