Hindawi
Wireless Communications and Mobile Computing
Volume 2023, Article ID 9809531, 1 page
https://doi.org/10.1155/2023/9809531

Retraction

Retracted: A Formal Verification Method of Compilation Based on C Safety Subset

Wireless Communications and Mobile Computing

Received 28 November 2023; Accepted 28 November 2023; Published 29 November 2023

This article has been retracted by Hindawi, as publisher, following an investigation undertaken by the publisher [1]. This investigation has uncovered evidence of systematic manipulation of the publication and peer-review process. We cannot, therefore, vouch for the reliability or integrity of this article.

Please note that this notice is intended solely to alert readers that the peer-review process of this article has been compromised.

Wiley and Hindawi regret that the usual quality checks did not identify these issues before publication and have since put additional measures in place to safeguard research integrity.

We wish to credit our Research Integrity and Research Publishing teams and anonymous and named external researchers and research integrity experts for contributing to this investigation.

The corresponding author, as the representative of all authors, has been given the opportunity to register their agreement or disagreement to this retraction. We have kept a record of any response received.

References

[1] Y. Tan, D. Ma, and L. Qiao, "A Formal Verification Method of Compilation Based on C Safety Subset," *Wireless Communications and Mobile Computing*, vol. 2021, Article ID 8352267, 10 pages, 2021.

Hindawi
Wireless Communications and Mobile Computing
Volume 2021, Article ID 8352267, 10 pages
https://doi.org/10.1155/2021/8352267

Research Article

A Formal Verification Method of Compilation Based on C Safety Subset

Yu Tan,[1,2] Dianfu Ma,[1,2] and Lei Qiao[3]

[1]*State Key Laboratory of Software Development Environment, Beihang University, Beijing 100191, China*
[2]*School of Computer Science and Engineering, Beihang University, Beijing 100191, China*
[3]*Beijing Institute of Control Engineering, Beijing 100190, China*

Correspondence should be addressed to Lei Qiao; 105133744@qq.com

Received 17 June 2021; Revised 3 July 2021; Accepted 12 July 2021; Published 1 August 2021

Academic Editor: Balakrishnan Nagaraj

With the rapid increase in the number of wireless terminals and the openness of wireless networks, the security of wireless communication is facing serious challenges. The safety and security of computer communication have always been a research hotspot, especially the wireless communication that still has a more complex architecture which leads to more safety problems in the communication system development. In recent years, more and more wireless communication systems are applied in the safety-critical field which tends to need high safety guarantees. A compiler is an important tool for system development, and its safety and reliability have an important impact on the development of safety-critical software. As the strictest method, formal verification methods have been widely paid attention to in compiler verification, but the current formal verification methods have some problems, such as high proof complexity, weak verification ability, and low algorithm efficiency. In this paper, a compiler formal verification method based on safety C subsets is proposed. By abstracting the concept of C grammar units from safety C subsets, the formal verification of the compiler is transformed into the verification of limited C grammar units. In this paper, an axiom system of first-order logic and special axioms are introduced. On this axiom system, the semantic consistency verification of C grammar unit and target code pattern is completed by means of theorem proving, and the formal verification of the compiler is completed.

1. Introduction

With the rapid development of wireless communication technology, a wireless communication system has been widely used in various fields of national economy and national defense construction. An important part of all kinds of safety-critical systems and software have more and more complex internal structures and open application environments. These factors make people pay more attention to its safety and reliability. Therefore, it is very important to analyze, design, and verify the safety-critical software, especially the onboard software of large passenger aircraft.

At present, the certification standard mainly adopted in the aviation field is DO-178B [1], "Software Requirements in the Certification of Airborne Systems and Equipment", issued by the American Aeronautical Radio Commission (RTCA) in December 1992. DO-178B defines the design and development process for airborne software and describes the target traceability process. According to the impact on aircraft failure state, airborne software is classified into five software grades: A (catastrophic), B (serious), C (heavy), D (light), and E (no impact). The DO-178C was released by RTCA in 2012 [2]. DO-178C complements DO-178B in four aspects: software tool verification, model-based development and verification, object-oriented programming, and formal methods. With the rapid development of new software technologies, these additions and revisions are well adapted to the current process of safety-related software development.

Compiler, as an important tool in software development process, is the bridge between software design and hardware operation. How to ensure the correctness of compiler compilation process is an important problem in software

development. The traditional method to detect compilation errors is to conduct a lot of tests, but the tests can only prove that the software is wrong but cannot prove that the software is error-free. In recent years, formal verification methods have received continuous attention in compiler verification. The formal verification method is based on strict mathematical theory, and the software system and properties are regulated by logical methods. Through the formal system composed of axioms and inference rules, the software system is proved in the same way as the theorem proved in mathematics. At present, there are two mainstream methods for formal verification of compilers. One is model detection [3], whose main bottleneck is state explosion, so it is only suitable for analysis of systems with limited and small state space. The other is theorem proving technology [4]. At present, a better way of proving is to use the unified framework of programming and proving, such as PVS, Coq, and Isabelle. Its limitation lies in that it requires tool users to have a high theoretical foundation and technical background, which limits the learning and application of most programmers.

This paper proposes a compiler formal verification method based on secure C subset. By introducing MISRA-C [5], the automotive manufacturing embedded C coding standard, the C language used in safety-critical systems is restricted. The C language defined by this specification is considered to be readable, reliable, portable, and easy to maintain. Combined with the characteristics of MISRA-C and aerospace software, a series of programming criteria of C language software are redefined, and a C safety subset is formed. The C safety subset strictly requires the maturity and stability of the compiler. The compiler must truly reflect the structure and semantics of the source code for comparison and tracking before and after compilation. From the C safety subset, a finite C grammar unit can be obtained, and the source program can be identified by the push-down automata corresponding to the grammar unit. A correct source program must conform to the rules of the grammar unit. Therefore, the proof of the correctness of the compilation process of the source program can be transformed into the proof of the equivalence of the semantics of the grammar unit. This method greatly simplifies the problems of the traditional formal verification methods, such as the high complexity and long time of the proof.

2. Related Works

The study of theorem proving for programs began in the 1960s with the papers published by Hoare and Floyd [6]. In his paper, Hoare proposed a formal system called Hoare logic. In Hoare logic, there is a set of proof rules called Hoare rules. Hoare logic can be used for machine proof because Hoare rule can be used for formal proof of some correctness assertions.

Separation logic is an extension of Hoare logic, which eliminates the possibility of sharing by providing logical connectives to express separation and corresponding deduction rules, and can describe the properties and related operations of memory in a natural way in the process of calculation, thus simplifying the verification of pointer programs. Separate logic has been proven to be more verifiable, which is a big step forward in program verification and inference technology.

In recent years, the representative one is the work of the COMPCERT project team led by Leroy X [7, 8]. For the first time, they have completed the formal verification of the correctness of a complete and practical compilation process. The whole verification process is completely formalized and automatically generated by the machine. To support automatic formal verification, CompCert uses the Coq Assistant, an auxiliary theorem proving tool, to restructure the compilation process. The entire process consists of conversions between eight different intermediate languages, and then, the Coq Assistant is used to prove the correctness of the entire compilation process, i.e., semantic retention. At present, the CompCert compiler can only realize compilation verification for a subset of C language, and it cannot fully cover all C language elements. Moreover, the back-end optimization degree is relatively low, and the project is still under further study [9].

In 2011, Yang et al. [10] tested mainstream C compilers in their research work on CSmith, an automatic test case generation tool, and reported 325 previously unknown bugs to compiler developers, including famous Intel CC, GCC, and LLVM compilers. CompCert performed exceptionally well in all 11 C compilers it compared, not finding any wrong-code errors in its supported C subsets.

The most influential work in recent years is that CompCert puts forward a safe subset Clight in the compiler researched and developed. By adding ultralong types, variable-length parameters, and other features, and adding a safe subset in the front end of the compiler to analyze and trim unnecessary language features in the front end, the compiler complexity of the language is reduced. The range between the standard C (C90), Clight, and MISRA-C safety subsets is shown in Figure 1.

As shown in Figure 1, the C90 covers the latter two, being the first standard C specification to add a true standard library, new preprocessing commands and features, function prototypes that allow for function declarations, and new keywords for specifying parameter types. There is a large intersection between Clight and MISRA-C, but some of them are not covered by each other. This has the very important instructive significance to the future work. Part 1 represents the C language features included in the declaration definition, layout writing, branch control, pointer use, jump control, operational processing, and other parts that are not supported by MISRA-C. Part 2: the intersection of the C language subset supported by Clight and MISRA-C, including C language basic types, syntax, statements, and library functions. Part 3: restricted multiple label declaration, null switch language use, parameter pointer use, pointer nesting, and so on stipulated by MISRA-C are included in it. Part 4: this region is the point not covered by the two subsets, and it is the C language features such as extralong data type, triplet sequence, special type of structure, general representation of floating-point type, special definition of precursor volume, and special arithmetic library function, which are clearly defined in C90 standard.

After the investigation of related researches, it is found that the research content of safety C compilation mainly includes two aspects: the correctness of the design and

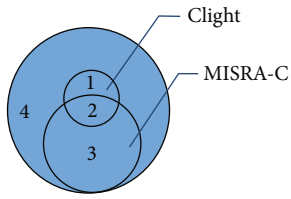

Figure 1: The scope of Clight, MISRA-C, and C90.

function of compilation, which ensures the correctness of the compiled object and the correctness of the compilation process itself, and thus guarantees the compilation security to a certain extent obviously; the security of the compiled object is the premise, and the security of the compiler itself is the ultimate goal. Both of them are purposes and motivations of each other, and together constitute the research contents of safety-critical compilation. The conclusions of the objects studied in this paper are shown in Table 1.

3. Formal Verification Method of Complication

3.1. Safety C Grammar Unit and Semantics. Abstract Syntax Tree Code (ASTC) is a tree representation of the abstract syntax structure of the source code, with each node on the tree representing a syntax structure in the source code. The source program and its corresponding abstract syntax tree are essentially equivalent, and the abstract syntax tree contains all the semantic information of the source program, so the verification of the source program can be transformed into the verification of the abstract syntax tree. From the definition of the tree, we can know that each tree is recursively defined by its subtree, so we can verify the whole grammar tree by verifying the subtree of the grammar separately

Under the constraints of the C safety subset, the unsafe syntax specifications in C language, such as Pointers, are removed. Abstract and induction of the abstract syntax tree, we can get some common subsyntax tree structure, these subsyntax tree structures were extracted and restored to the form of C language, and we get the C grammar unit. Compilation verification of the source program is equivalent to verification of each C grammar unit, which can be achieved by verifying whether the semantics of each C grammar unit and the corresponding target code pattern are consistent before and after compilation.

In order to obtain the semantics of each C grammar unit, this paper introduces the concept of context, which can define the semantics of grammar unit according to context. Context represents the environment and context of each proof item in the sequence to be proved, including function local variables, global variables, and context. Table 2 shows some of the C grammar units and their corresponding semantics.

In the table, the sigma symbol σ represents the process of taking a value, and σ(<LOG-EXP>) represents the value of a logical expression that, according to the C safety subset specification, can only be 0 and 1. <STA-LIST> represents a statement block, which can include expression statement and conditional selection statement, and is generally handed over to the push-down automaton that identifies the statement block for recursive processing. <ASS-EXP> represents an assignment statement that returns the value of the expression. "{..} ∗∗ n" represents the statement inside the braces of the loop execution, which defines the semantics of the loop statement. SKIP means to jump directly to the next statement for execution. Under the 32-bit Power PC instruction set, define SKIP equals $\sigma(\mathrm{PC} = \mathrm{PC} + 4)$, where PC represents the program counter.

3.2. The Denotational Semantics of Target Language. The denotational semantics is the science of annotating the semantics of a definite formal language with corresponding mathematical objects (such as set and function) using the formal system method. The denotational semantics can also be explained as follows: there are two domains, one is the linguistic domain, in the grammatical domain defines a formal language system. The other is the mathematical domain (or formal system with known semantics). Within the scope of a 32-bit Power PC instruction set, this paper models assembly instruction and obtains corresponding the denotational semantics according to the operational semantics of each instruction given in the official Power PC document. Table 3 gives the denotational semantics of some Power PC assembly instructions.

In the table, the GPR (general-purpose register) represents the Power PC's general-purpose register, mainly used as a stack pointer, the first argument to a function, the return value, etc. CR (Conditional Register) is a condition register, which can reflect the results of some operations (such as CMP instruction) and assist in testing and execution of branch instructions. MEM is a memory space that stores the values of local, global, and other variables. @target represents the relative address, which is generally used in the jump instruction. PC = PC + @target means to jump from the address currently pointed by PC to the address identified by the target. PC = PC + 4 means to directly execute the next instruction.

3.3. Target Code Patterns and Propositions. Target code pattern is a generalized representation of target code sequence obtained by compiling C grammar units in a certain context with GCC compiler and eliminating the influence of context on the target code sequence. Formal verification of compiler should be converted into semantic consistency verification of C grammar unit and target code pattern. Table 4 shows the target code patterns corresponding to some C grammar units under the 32-bit Power PC instruction set.

Formal proof of a program requires a specific system of axioms as the basis. The axiomatic system is to axiomatize a scientific theory and study it with axiomatic methods. Every scientific theory is a system composed of a series of concepts and propositions. Based on the referential semantics of Power PC assembly instruction in 2.2, the proposition mapping algorithm in 3.1 can be used to obtain the proposition of each target code pattern. The propositions of target code patterns about the condition statement (<if-statement>) and loop statement (<while-statement>) are given in Table 5.

3.4. Verification and Proof. The formal verification method proposed in this paper is based on the axiom system of

Table 1: Comparisons of related works.

Works and method		Advantages	Deficiency
Correctness of design	C safety subset	A safe subset is proposed, which can reduce the complexity of compilation by adding extralong types, variable-length parameters, and other features in the front end without changing the C semantics.	The coverage range is smaller than that of C90 standard, and MISRA-C standard is not covered, and the accuracy of detection and analysis is not complete.
Correctness of function	From C to ASM	It supports automatic formal proof and reconstructs the compilation process in Coq and completes the compilation from a structured functional language Clight to assembly code PowerPC. Finally, it proves the correctness of the whole compilation process by using Coq.	In the process, it has experienced compilation between eight different intermediate languages, and the language changes are complex. The project is still in progress.
Formal verification	Model checking	High degree of automation, error detection can be given after the counterexample.	State space explosion problem.
	Theorem proving	Using rigorous mathematical reasoning to prove clear attribute requirements and clear correctness standards, high credibility.	The degree of automation is low, the degree of formalization requirements is high, and the proof process is difficult.

Table 2: Safety C grammar unit and semantics.

Statement	Grammar unit	Grammar unit semantics

<if-statement>	if (<LOG-EXP>)
	{
	<STA-LIST_1>
	}
	else
	{
	<STA-LIST_2>
	}
		σ(<LOG-EXP>) -> σ(<STA-LIST_1>)
		~σ(<LOG-EXP>) -> σ(<STA-LIST_2>)

<while-statement>	while (<LOG-EXP>)
	{
	<STA-LIST>
	}
		{σ(<LOG-EXP>) -> σ(<STA-LIST>)} ** n
		~σ(<LOG-EXP>) -> skip

<do-while-statement>	do { <STA-LIST> } while (<LOG-EXP>);	σ(<STA-LIST>) {σ(<LOG-EXP>) -> σ(<STA-LIST>)} ** n ~σ(<LOG-EXP>) -> skip

<for-statement>	for(<ASS-EXP_1>;
	<LOG-EXP>;
	<ASS-EXP_2>)
	{
	<STA-LIST>
	}
		σ(<ASS-EXP_1>)
		{σ(<LOG-EXP>) -> σ(<STA-LIST>);
		σ(<ASS-EXP_1>)} ** n
		~σ(<LOG-EXP>) -> skip

first-order logic. Starting from the axioms given in the axiom system in advance (such as the target code pattern proposition), a series of new propositions are deduced according to the inference rules and are added as the premise in the subsequent proof process. Since every item in the proof sequence is a premise, an axiom, or a theorem, and because the axiom system of first-order logic is reliable, every item in the proof sequence must be correct, so that the final inference proves the sequence must be correct.

The MP (Modus Ponens) rule, which is called detached argument or detached rule, is the most basic reasoning rule in the axiom system of first-order logic. The MP rule can be expressed as follows:

$$p \longrightarrow q, p \vdash q. \tag{1}$$

The implication is as follows: if p is true, then q is true, and then p is true, therefore q is true. The separation rule consists of three statements (or propositions, or statements): the first statement is a conditional statement, i.e., p implicates q; the second statement is p, the premise of the conditional statement is true. The q follows from the first two statements that the conclusion of the conditional statement must also be true logically.

The CI (Conjunction Introduction) rule is called the Conjunction Introduction or combination rule. CI rule can be expressed as follows:

Table 3: The denotational semantics.

Instruction category	Instruction	Denotational semantics
li	li rD,SIMM	GPR[rD] = SIMM
lwz	lwz rD, D(rA)	GPR[rD] = MEM[D]
stw	stw rS,D(rA)	MEM[D] = GPR[rS]
b	b target	PC = PC + @target
beq	beq crfD,target	CR[crfD] == b100->PC = PC + 4 CR[crfD] == b010->PC = PC + 4 CR[crfD] == b001->PC = PC + @target
bne	bne crfD,target	CR[crfD] == b100->PC = PC + @target CR[crfD] == b010->PC = PC + @target CR[crfD] == b001->PC = PC + 4
cmp	cmp crfD,L,rA,rB	GPR[0] < 0->CR [7] = b100 GPR[0] > 0->CR [7] = b010 GPR[0] == 0->CR [7] = b001
cmpi	cmpi crfD,L,rA,SIMM	GPR[rA] < SIMM->CR[crfD] = b100 GPR[rA] > SIMM->CR[crfD] = b010 GPR[rA] == SIMM->CR[crfD] = b001
add	add rD,rA,rB	GPR[rD] = GPR[rA] + GPR[rB]
addic	addic	GPR[rD] = GPR[rA] + SIMM
subf	subf rD,rA,rB	GPR[rD] = -GPR[rA] + GPR[rB]
mullw	mullw rD,rA,rB	GPR[rD] = GPR[rA] ∗ GPR[rB]
divw	divw rD,rA,rB	GPR[rD] = GPR[rA]/GPR[rB]

$$p, q \vdash p \wedge q. \tag{2}$$

Its meaning is as follows: if p, q is true, then $p \wedge q$ is true. The rule of conjunction is mainly used to convert multiple true propositions into a single one.

In the actual verification process, for the target code mode propositions such as expression grammar unit and conditional choice grammar unit, because they do not contain cyclic structure, it is convenient to complete the reasoning proof of propositions by using MP rule and axiom set and theorem set in the axiom system of first-order logic. However, for cyclic structure, such as <While-Statement> target code pattern proposition, after direct reasoning, the semantics of the target code pattern obtained is quite different from the semantics of the C grammar unit, so it is impossible to directly prove that the semantics of the two are consistent. Therefore, this paper introduces the limited mathematical induction method to prove the target code pattern proposition of cyclic structure.

The logical basis of limited mathematical induction is the axiom of natural numbers, also known as Peano's axiom. General mathematical induction logic expressions for the $P(0) \bigwedge (\forall n)(P(n) \longrightarrow P(s(n))) \longrightarrow (\forall n) P(n)$, limited mathematical induction is on the basis of general mathematical induction, limit n is poor, the program for loop structure, the loop is terminated, and termination conditions are given by the people. Table 6 and Algorithm 1 will give the proofs of the target code pattern propositions of <if-statement> and <while-statement>, respectively.

In Table 6, the final inference of the proof sequence is S12, and the semantics of the target code pattern obtained by the value (σ) operation on S12 are as follows:

$$\sigma(\text{<LOG-EXP >}) - > \sigma(\text{<STA-LIST}_1\text{ >}) \| \sim \sigma(\text{<LOG-EXP >}) \longrightarrow \sigma(\text{<STA-LIST}_2\text{ >}). \tag{3}$$

Combined with the semantics of <if-statement> in Algorithm 1, it can be seen that the semantics of the two are consistent, which is verified.

The inference of <while-statement> proof sequence is the same as that of <if-statement>, the proof sequence is directly derived here, and the proof sequence is proved by finite mathematical induction combining with the semantics of the grammar unit.

4. Formal Verification Algorithm

4.1. Proposition Mapping Algorithm. The function of the propositional mapping algorithm is to transform the object code pattern into the form of propositions, so as to facilitate subsequent reasoning proof. The algorithm needs to take the referential semantics corresponding to each instruction in the Power PC instruction set as the special axiom input, traverse the input object code pattern one by one, transform each object code into the corresponding form of referential semantics, and finally express the referential semantics of the object code pattern into the form of proposition set

Table 4: Grammar unit and patterns of target code.

Statement	C grammar unit	Target code pattern

<if-statement>	if (<LOG-EXP>)
	{
	<STA-LIST_1>
	}
	else
	{
	<STA-LIST_2>
	}
		<LOG-EXP>
		cmpi 7,0,0,0
		beq 7,.L1
		<STA-LIST_1>
		b .L2
		.L1;
		<STA-LIST_2>
		.L2:

<while-statement>	while (<LOG-EXP>)
	{
	<STA-LIST>
	}
		b .L2
		.L1:
		<STA-LIST>
		.L2:
		<LOG-EXP>
		cmpi 7,0,0,0
		bne 7,.L1

<do-while-statement>	do { <STA-LIST> } while (<LOG-EXP>);	.L1: <STA-LIST> <LOG-EXP> cmpi 7,0,0,0 bne 7,.L1

<for-statement>	for(<ASS-EXP_1>;
	<LOG-EXP>;
	<ASS-EXP_2>)
	{
	<STA-LIST>
	}
		<ASS-EXP_1>
		b .L2
		.L1:
		<STA-LIST>
		<ASS-EXP_2>
		.L2:
		<LOG-EXP>
		cmpi 7,0,0,0
		bne 7,.L1

Table 5: The propositions of target code patterns.

Statement	Target code pattern	Proposition
<if-statement>	<LOG-EXP> cmpi 7,0,0,0 beq 7,.L1 <STA-LIST_1> b .L2 .L1: <STA-LIST_2> .L2:	P1: GPR[0] = <LOG-EXP > P2: (GPR[0] < 0->CR[7] = b100) ‖ (GPR[0] > 0->CR[7] = b010) ‖ (GPR[0] == 0->CR[7] = b001) P3: (CR[7] == b100->PC = PC + 4) ‖ (CR[7] == b010->PC = PC + 4) ‖ (CR[7] == b001->PC = PC + @.L1) P4 = <STA-LIST_1 > P5: PC = PC + @.L2 P6: .L1: P7: <STA-LIST_2> P8: .L2:
<while-statement>	b .L2 .L1: <STA-LIST> .L2: <LOG-EXP> cmpi 7,0,0,0 bne 7,.L1	P1: PC = PC + @.L2 P2: .L1: P3: <STA-LIST> P4: .L2: P5: GPR[0] = <LOG-EXP > P6: (GPR[0] < 0->CR[7] = b100) ‖ (GPR[0] > 0->CR[7] = b010) ‖ (GPR[0] == 0->CR[7] = b001) P7: (CR[7] == b100->PC = PC + @.L1) ‖ (CR[7] == b010->PC = PC + @.L1) ‖ (CR[7] == b001->PC = PC + 4)

Table 6: The proof of <if-statement>.

Proof	Inference
S1 = GPR[0] = <LOG-EXP >	P1
S2 = (GPR[0] < 0->CR [7] = b100) ‖ (GPR[0] > 0->CR [7] = b010) ‖ (GPR[0] == 0->CR [7] = b001)	P2
S3 = (<LOG-EXP><0->CR [7] = b100) ‖ (<LOG-EXP>>0->CR [7] = b010) ‖ (<LOG-EXP> == 0->CR [7] = b001)	S1, S2, MP
S4 = (CR [7] == b100->PC = PC + 4) ‖ (CR [7] == b010->PC = PC + 4) ‖ (CR [7] == b001->PC = PC + @.L1)	P3
S5 = (<LOG-EXP><0->PC = PC + 4) ‖ (<LOG-EXP>>0->PC = PC + 4) ‖ (<LOG-EXP> == 0->PC = PC + @.L1)	S3, S4, MP
S6 = <STA-LIST_1 >	P4
S7 = PC = PC + @.L2	P5
S8= .L1:	P6
S9 = <STA-LIST_2 >	P7
S10= .L2:	P8

S11 = {
(<LOG-EXP><0->PC = PC + 4) ‖ (<LOG-EXP>>0->PC = PC + 4) ‖ (<LOG-EXP> == 0->PC = PC + @.L1) ∧
<STA-LIST_1 > ∧
PC = PC + @.L2 ∧
.L1: ∧
<STA-LIST_2 > ∧
.L2:
}
	S5, S6, S7, S8, S9, S10, CI

S12 = {
(<LOG-EXP><0-><STA-LIST_1 >) ‖ (<LOG-EXP>>0-><STA-LIST_1 >) ‖ (<LOG-EXP> == 0–><STA-LIST_2 >)
}
	S11

Proposition:$\{\sigma(<LOG-EXP>)->\sigma(<STA-LIST>)\} **n\|\sim \sigma(<LOG-EXP>)->$ skip

Lemma:PC = PC + @.L2 ∧
.L1: ∧
<STA-LIST> ∧
.L2: ∧
(<LOG – EXP><0–>PC = PC + @.L1)‖(<LOG – EXP>>0–>PC = PC + @.L1)‖(<LOG – EXP> == 0–>PC = PC + 4)

Proof:

(1) when n = 1, substitution propositions are: $\sigma(<LOG-EXP>)->\sigma(<STA-LIST>)\|\sim \sigma(<LOG-EXP>)->$skip.

To lemma, when n is 1, said only cycle time, use CI rules: (<LOG – EXP><0–><STA – LIST >)‖(<LOG – EXP>>0–><STA – LIST >)‖(<LOG – EXP> == 0–>PC = PC + 4),

Values, available lemma semantics: (<LOG – EXP><0–><STA – LIST >)‖(<LOG – EXP>>0–><STA – LIST >)‖(<LOG – EXP> == 0–>PC = PC + 4),

It can be obtained that the semantics of the two are consistent, so it is true when k = 1.

(2) Assume n = n, get the proposition $\{\sigma(<LOG-EXP>)->\sigma(<STA-LIST>)\} **N\|\sim \sigma(<LOG-EXP>)->$skip was established.

When n = n + 1, on the basis of n = n, a cycle is performed.

If <LOG – EXP > == 0, then PC = PC + 4, the whole loop is finished, and the semantics obtained after value operation are:

$\sim \sigma(<LOG-EXP>) -> $ skip.

If <LOG – EXP > ! = 0, then PC jumps to the starting position of <STA-LIST>, continues to execute the statement sequence, and the semantics obtained after value operation are:

$\sigma(<LOG-EXP>)->\sigma(<STA-LIST>)$,

Apply the semantic and (2) the above assumptions, CI rules, have $\{\sigma(<LOG-EXP>)->\sigma(<STA-LIST>)\} **(N+1)\|\sim \sigma(<LOG-EXP>)->$ skip was established.

From (1), (2) and (3), it can be seen that the propositions to be verified are valid and have been verified.

Algorithm 1: The proof of <while-statement>.

```
Input: TargetCodePatternSet
Output: PropositionSet
1: axiomSet = loadAxiom(denotationalSemanticsFileName)
2: for each line in TargetCodePatternSet do
3:    lines = line.split(regex)
4:    lines = filterOtherCharacter(lines)
5:    if lines.length ==0 then
6:       continue
7:    else if lines.length == 1 then
8:       add new Proposition(lines) to PropositionSet
9:    else
10:       paras = generateParas(lines)
11:       seman = generateSemantic (lines, paras, axiomSet)
12:       add new Proposition (lines, paras, seman) to PropositionSet
13:    end if
14: end for
```

Algorithm 2: Proposition mapping algorithm.

```
Input:PropositionSet
Output: SemantemeSet
1:for each p in PropositionSet do
2:    for each q in PropositionSet do
3:       if p == q then
4:          continue
5:       end if
6:       newProposition = applyInferenceRuleToTwoPropositions (p, q)
7:       if newProposition != null then
8:          add newProposition to PropositionSet
9:       end if
10:       if q's content is empty then
11:          remove q from PropositionSet
12:       end if
13:    end for
14: end for
15: for each p in PropositionSet do
16:    s = obtainSemantemeFromProposition (p)
17:    add s to SemantemeSet
18: end for
```

Algorithm 3: Automatic inference algorithm.

output. The pseudocode of the propositional mapping algorithm is shown in Algorithm 2.

4.2. Automatic Inference Algorithm. The automatic inference algorithm is the core of the formal verification method proposed in this paper. The algorithm takes the proposition set outputted by the proposition mapping algorithm in 3.1 and the axiom set of first-order logic as the premise, deduces a series of new propositions according to the inference rules, adds these new propositions to the premise for subsequent proof, and finally constructs the proof sequence and draws the conclusion. For the object code pattern propositions without cycles, we can directly value (σ) the sequence of the proof, so as to obtain the semantics of the object code pattern propositions. Compared with the semantics of C grammar units, we can directly judge whether the semantics of the two are consistent. For the object code pattern proposition set containing cycles, the semantic consistency verification process cannot be completed by directly valuing the derived proof sequence, and the loop interactive proof algorithm in 3.3 can be introduced to complete the whole process. The pseudocode of the automatic reasoning algorithm for propositions is shown in Algorithm 3.

4.3. Loop Interaction Proof Algorithm. The theoretical basis of the loop interactive proof algorithm is limited mathematical induction. The algorithm first guide the user to enter n to 1 C unit of semantics, grammar, and then according to the loop condition is true or false, respectively, construct new proposition to join proposition 3.1 mapping algorithm output a

```
Input: PropositionSet
Output: Flag
1: Flag = true
2: for i from 1 to 2 do
3:   userSemantemeSet = ReadUserInputSemanteme ()
4:   copy PropositionSet to cpyPropositionSet
5:   add true loop condition Propositionto cpyPropositionSet
6:   trueSemantemeSet = AutomaticInferenceAlgorithm(cpyPropositionSet)
7:   copy PropositionSet to cpyPropositionSet
8:   add false loop condition Propositionto to cpyPropositionSet
9:   falseSemantemeSet = AutomaticInferenceAlgorithm(cpyPropositionSet)
10:      semantemeSet = trueSemantemeSet || falseSemantemeSet
11:  if i == 2 then
12:        semantemeSet=CI (semantemeSet, userSemantemeSet)
13:        update n from N to (N + 1) in userSemantemeSet
14:  end if
15:  if semantemeSet unequal to userSemantemeSet then
16:        Flag = false
17:        return Flag
18:  end if
19: end for
```

Algorithm 4: Loop interaction proof algorithm.

copy of the thesis set, call the automated reasoning algorithm to copy proposition 3.2 set reasoning, so as to get the target pattern proposition of semantics.

Contrast the semantics of object code pattern propositions with the semantics of user input. If the semantics of the two are inconsistent, it will give a direct reminder of the errors in the formal verification process and exit. If it is consistent, the user is reminded to input the semantics of the C grammar unit when n is N, and on this basis, the source semantic set is reasoned again. Through CI rules, the inferred semantic results are added to the semantics of the C grammar unit when n is N, and the semantics of the target code pattern when n is $N + 1$ are obtained. When the user input n is N, the semantics of the C grammar unit is replaced by $N + 1$, and the target code pattern semantics inferred by the program is compared with the semantics of the user input C grammar unit when n is $N + 1$, and the judgment result is returned. The pseudocode of the loop interaction proof algorithm is shown in Algorithm 4.

5. Discussion and Conclusion

In the safety C to the target code compilation process, different scenarios in the field of safety critical need to support more kinds of assembly instruction set, considering the method of this paper can be generalized to a variety of platforms, and the next step works consideration to expand to the academic and industry mainstream assembly instruction set, in order to better support security system application in key areas. This part of the work has been carried out in the laboratory where this paper is located. Up to now, it can support MIPS, x86, SPARC, etc. And RISC-V, the fifth generation of open source condensed instruction set, is also under research.

Also, optimization is a common method to improve the compilation efficiency, but the introduction of optimization will cause some details missing in the compilation process, making it difficult to verify the traceability of source code and object code. How to guarantee both compilation systems meet the demand of traceability in the field of security key, and the optimization technique in safety-critical software compilation; this paper is preliminary thought to compile step for progressive refinement, the architecture, development, and validation of refinement to enhance the analysis of particle size; it will be the issues worthy of challenges in the future research.

This paper proposes a safety subset of the C compiler form verification method towards wireless communication systems which are applied in the safety-critical field; this method is within the scope of C safety subset constraints C grammar unit which is introduced, the traditional directly to the source code of the overall transformation of formal verification in order to limit C verification of grammatical unit, by showing that C grammar unit and the target pattern of semantic equivalence completed form verification process. This method greatly reduces the complexity and time of formal verification and saves the development cost. At the same time, the verification tool developed based on the formal verification algorithm proposed in this paper can guide the user to complete the proof of the cyclic structure through very few interactions, and the whole proof process conforms to the principle of restricted mathematical induction.

The future work will focus on further refinement of C grammar unit and object code pattern, and further research on cyclic interaction proof algorithm will be needed, so that the algorithm can complete the reasoning process with less or even no artificial interaction.

Data Availability

The datasets used and/or analyzed during the current study are available from the corresponding author on reasonable request.

Conflicts of Interest

The authors declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Authors' Contributions

Yu Tan drafted the manuscript, Dianfu Ma participated in the design of the study and performed the statistical analysis, and Lei Qiao conceived the study and participated in its design and coordination. All authors read and approved the final manuscript. All contributors who do not meet the criteria for authorship should be listed in an acknowledgements section.

Acknowledgments

The authors acknowledge the Natural Science Foundation of China (Grant: NSFC 61632005 and 62032004).

References

[1] G. Zoughbi, L. Briand, and Y. Labiche, "Modeling safety and airworthiness (rtca do-178b) information: conceptual model and uml profile," *Software & Systems Modeling*, vol. 10, no. 3, pp. 337–367, 2011.

[2] W. K. Youn, S. B. Hong, K. R. Oh, and O. S. Ahn, "Software certification of safety-critical avionic systems: do-178c and its impacts," *Aerospace & Electronic Systems Magazine IEEE*, vol. 30, no. 4, pp. 4–13, 2015.

[3] N. Chondamrongkul, J. Sun, I. Warren, and S. U. Lee, "Integrated formal tools for software architecture smell detection," *International Journal of Software Engineering and Knowledge Engineering*, vol. 30, no. 6, pp. 723–763, 2020.

[4] N. Courant and X. Leroy, "Verified code generation for the polyhedral model," *Proceedings of the ACM on Programming Languages*, , no. POPL, 2021.

[5] G. Mauro, H. Thimbleby, A. Domenici, and C. BeRnardeschi, "Extending a user interface prototyping tool with automatic misra c code generation," *Electronic Proceedings in Theoretical Computer Science*, vol. 240, pp. 53–66, 2017.

[6] T. Sun and W. Yu, "A formal system of axiomatic set theory in coq," *IEEE Access*, vol. 8, pp. 21510–21523, 2020.

[7] T. Bourke, L. Brun, P. É. Dagand, X. Leroy, and L. Rieg, "A formally verified compiler for Lustre," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 586–601, 2017.

[8] Y. K. Tan, M. O. Myreen, R. Kumar, A. Fox, S. Owens, and M. Norrish, "The verified cakeml compiler backend," *Journal of Functional Programming*, vol. 29, 2019.

[9] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," *ACM SIGPLAN Notices*, vol. 51, no. 10, pp. 849–846, 2016.

[10] M. Rabin and M. A. Alipour, "Configuring test generators using bug reports: a case study of gcc compiler and csmith," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing–Software Verification and Testing Track (SAC-SVT'21)*, 9 pages, ACM, New York, NY, USA, 2020.