

# SCALC: A Safety-Case-Guided Assurance Framework for Lustre Compilers

Shurong Shi<sup>1</sup>, Yuxin Deng<sup>2,1\*</sup>, Yanhong Huang<sup>1</sup>, and Wenjie Du<sup>3</sup>

<sup>1</sup> East China Normal University, China

51285902099@stu.ecnu.edu.cn, yhhuang@sei.ecnu.edu.cn

<sup>2</sup> Shanghai University of Finance and Economics, China

yxdeng@msg.sufe.edu.cn

<sup>3</sup> Shanghai Normal University, China

wenjiedu@shnu.edu.cn

**Abstract.** Compilers for synchronous dataflow languages such as Lustre are critical components in the development of safety-critical embedded systems. Although formal verification has been applied to several Lustre compilers, practical assurance remains challenging because verified translation stages may rely on incomplete semantic specifications, while unverified front-end stages can still introduce defects. Consequently, there is still no unified testing and assurance framework specifically designed for Lustre compilers. We present SCALC, a safety-case-guided assurance framework for Lustre compiler validation. SCALC uses structured safety cases to derive assurance goals, select validation activities, and organize the resulting evidence, integrating four complementary techniques: LLM-assisted generation of test programs targeted at specific assurance goals, manual review of translation rules, differential testing between independently developed Lustre compilers, and bounded functional equivalence checking of generated C code. We evaluate SCALC on the open-source L2C compiler and identify seven defects and two unsupported language features across multiple translation stages, including one defect in a Coq-verified part of the pipeline. These results show that SCALC can expose practical assurance gaps in partially verified compilers and provide traceable evidence for improving compiler reliability in safety-critical domains.

**Keywords:** Formal Methods · Lustre Compiler · L2C · Safety Case · Manual Review · Differential Testing

## 1 Introduction

The increasing deployment of embedded systems in real-time and safety-critical domains has intensified the demand for reliable software development methods. Lustre [9, 17] is a synchronous dataflow language widely used for modeling and implementing real-time embedded systems. Its temporal flow operators and deterministic execution model make it suitable for safety-critical applications. For example, Winandy et al. [36] use Lustre to model guidance, navigation, and control systems for space missions, while Lan

---

\* Corresponding author.

et al. [24] use it to describe control algorithms for safety-critical instrumentation and control systems in nuclear power plants. Lustre has also been used in runtime verification frameworks, such as the NASA-oriented approach of Perez et al. [30], where monitors are generated from natural-language requirements.

Currently, compilers for the Lustre language are rare, with notable examples including SCADE [13], Vélus [6], and L2C [32]. Among them, Vélus and L2C are open-source compilers that translate Lustre programs into C code and use Coq [1, 2] to verify critical translation steps. Despite these efforts, existing proofs do not establish whole-compiler reliability because they leave two technical gaps. A *coverage gap* remains when verified stages exclude front-end processing or unsupported language features. A *specification gap* remains because a proof establishes correctness only with respect to its formalized semantics, which may omit or misstate the intended Lustre behavior. Additional validation can reveal such problems, but isolated activities do not show which compiler-level claims have been examined or where evidence is still missing. This motivates a safety-case-guided approach: the safety case decomposes whole-compiler reliability into reviewable claims and requires explicit evidence for each claim, making both supported claims and open assurance gaps visible.

To address these gaps, we propose SCALC, a safety-case-guided assurance framework for Lustre compilers. SCALC integrates four complementary techniques to provide explicit evidence for whole-compiler reliability. To systematically supply the required validation inputs, an LLM-assisted mechanism builds upon Trusta [11] to generate test programs directly aligned with specific assurance goals. Manual translation-rule review compares the intended semantics of Lustre constructs with the semantics embodied in their translated representations. It can therefore reveal semantic misunderstandings that may have been incorporated into the formalized translation rules and would remain invisible to a proof based on those rules. Differential testing and bounded functional equivalence checking provide an external behavioral cross-check between two independently developed compilers. Differences in compilation behavior can expose defects in unverified front-end stages. Comparing the I/O behavior of the generated programs then provides evidence for assessing the formalized translation semantics. Differential testing supplies evidence over executed inputs, and bounded functional equivalence checking strengthens it for all inputs within the selected horizon. The safety case links these complementary results to their assurance claims and makes the remaining assumptions and unsupported goals explicit.

In the evaluation, SCALC identified seven distinct defects and two unsupported language features in the open-l2c-1.0 version of L2C. These issues span multiple translation stages, including both unverified early stages and one defect located in a Coq-verified part of the pipeline. This result shows that practical compiler assurance requires evidence beyond isolated translation proofs. Among the seven defects we identified, one stems from differences between L2C’s custom syntax and the official Lustre syntax, one has been fixed in a newer version of L2C that has not yet been open-sourced, and the remaining five have been confirmed by the L2C development team.

The main contributions of this paper, integrated within the proposed SCALC framework, include:

- **An LLM-assisted technique for generating safety-case-aligned test programs:** We extend Trusta [11], a safety-case construction tool, with a mechanism that uses LLMs to generate Lustre test programs from safety-case goals. This provides supplementary test inputs that are explicitly linked to assurance objectives.
- **A set of structured guidelines for manual review:** We outline a scope for the manual audit of Lustre compilers, organized in the form of safety cases. These structures provide hierarchical checklists to assist reviewers in inspecting compiler translation rules and syntactic coverage.
- **A hybrid methodology for differential testing of Lustre compilers:** We propose a differential testing methodology to identify behavioral inconsistencies between two Lustre compilers that generate C code. This methodology systematically compares a target compiler (L2C) against a reference compiler (Vélus) by using a combination of symbolic execution and fuzzing to test the input/output equivalence of the generated programs.
- **A method for bounded functional equivalence checking between C code generated by Lustre compilers:** This method applies bounded model checking and constraint solving directly to the C code generated by two Lustre compilers to check functional equivalence within a selected execution horizon.

The rest of the paper is structured as follows. Section 2 provides the background, and Section 3 details the LLM-based test generation and top-level safety case. Sections 4, 5, and 6 present the three main components of SCALC: manual review, differential testing, and bounded functional equivalence checking. Section 7 provides an experimental evaluation, detailing the identified defects and comparing the characteristics of the assurance methods. Finally, Section 8 discusses related work, and Section 9 concludes the paper.

## 2 Preliminaries

### 2.1 The Lustre Language

Lustre is a declarative synchronous dataflow language that has been widely used in real-time embedded systems, particularly in safety-critical domains. A Lustre program typically consists of nodes and declarations. A Lustre program contains type, constant, function, and node declarations. A node declares its inputs, outputs, and local variables, and its body contains equations that define the dataflow. Notable features of Lustre include the following:

- **Logical clocks:** Lustre programs operate over a discrete logical base clock, which does not by itself prescribe a fixed physical execution period. Each stream is associated with a clock and takes a value only at the logical instants when that clock is active; at other base-clock instants, the stream is absent. Derived clocks are typically subclocks of the base clock.
- **Synchronicity:** All calculations in the program are synchronous, completed in infinitesimal time. The program’s execution time is shorter than the interval between two base clock cycles.

**Table 1.** Core Lustre operators and their explicit dataflow semantics

| Operator             | Description                                                       | Input Stream(s)                                                        | Output Stream ( $O$ )                           |
|----------------------|-------------------------------------------------------------------|------------------------------------------------------------------------|-------------------------------------------------|
| $\text{pre } A$      | Returns the value of $A$ from the previous clock cycle.           | $A = (a_0, a_1, a_2, a_3, \dots)$                                      | $O = (\text{nil}, a_0, a_1, a_2, \dots)$        |
| $A \rightarrow B$    | Yields $A$ in the first cycle, and $B$ in all subsequent cycles.  | $A = (a_0, a_1, a_2, a_3, \dots)$<br>$B = (b_0, b_1, b_2, b_3, \dots)$ | $O = (a_0, b_1, b_2, b_3, \dots)$               |
| $A \text{ fby } B$   | Outputs $A$ initially, then delays $B$ by one cycle.              | $A = (a_0, a_1, a_2, a_3, \dots)$<br>$B = (b_0, b_1, b_2, b_3, \dots)$ | $O = (a_0, b_0, b_1, b_2, \dots)$               |
| $A \text{ when } ck$ | Samples $A$ only when the boolean clock $ck$ is true.             | $A = (a_0, a_1, a_2, a_3, \dots)$<br>$ck = (T, F, F, T, \dots)$        | $O = (a_0, \text{nil}, \text{nil}, a_3, \dots)$ |
| $\text{current } X$  | Expands a sampled stream $X$ by repeating its last non-nil value. | $X = (x_0, \text{nil}, \text{nil}, x_3, \dots)$                        | $O = (x_0, x_0, x_0, x_3, \dots)$               |
| $\text{not } A$      | Performs element-wise logical negation.                           | $A = (T, F, F, T, \dots)$                                              | $O = (F, T, T, F, \dots)$                       |

- Rich temporal operators: The Lustre language’s features are strongly time-related and provide many temporal operators. These can be used to record variable values from previous clock cycles and can also handle multi-rate systems, supporting the synchronization of data flows at different frequencies.

Table 1 provides an overview of the core Lustre operators and their dataflow semantics with explicit input and output stream examples.

## 2.2 L2C Compiler and Vélus Compiler

The L2C compiler, developed by Shang et al. [32], translates programs written in Lustre V6 into C with the goal of implementing a highly trustworthy compiler suitable for safety-critical systems.

Figure 1 illustrates the core pipeline, which progressively translates Lustre code into C through a series of intermediate representations. After lexical and syntactic analyses yield an AST, the compiler performs type and clock checking to generate the initial intermediate representation, LustreV. The pipeline then lowers this representation by eliminating high-level abstractions across several intermediate stages. During the transition through LustreS and LustreR toward LustreF, high-level constructs such as iterators are unrolled and temporal operators are normalized. The compilation proceeds with structure generation and variable separation, resulting in LustreE, LustreD, and LustreC. Finally, the representation is translated into Clight. Notably, L2C leverages Coq to prove the correctness of the translation process from LustreS to Clight.

The Vélus compiler [6] also employs Coq for formal verification. While it shares initial parsing and checking steps with L2C, its pipeline is more streamlined, converting synchronous dataflow constructs into an imperative language called Obc before generating Clight. Vélus is faster due to fewer translation stages [19].

## 2.3 Safety Cases

A safety case [3, 5] provides a systematic argumentation to demonstrate that systems meet safety and reliability requirements in specific scenarios. In practical terms, a safety

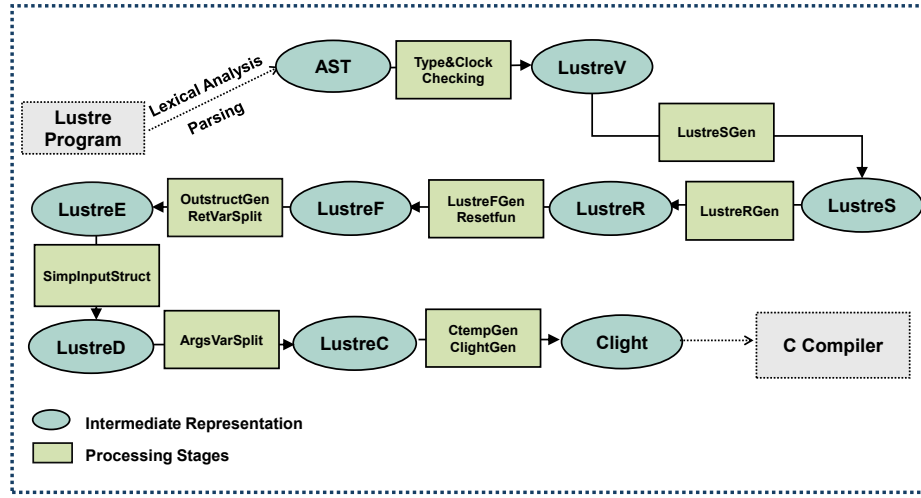


Fig. 1. Core translation process of L2C

case gives structured answers to three questions: What exactly is being claimed? Why should the claim follow from smaller, more specific claims? What concrete evidence supports those claims? It is therefore an explicit claim–argument–evidence structure rather than a safety proof by itself.

Safety cases include three main components: goals, arguments, and evidence. A *goal* is a claim that must be supported. An *argument* explains how a broad goal is decomposed into smaller obligations and why satisfying those obligations supports the broader claim. *Evidence* is an inspectable artifact, such as a review record, a test report containing concrete inputs and outputs, or a formal-verification result. A safety case makes the connection between each goal and its evidence visible, so that unsupported goals and assumptions can be identified during review. In this work, we adopt the Trustworthiness Derivation Tree (TDT) [14] for the formal representation of safety cases.

**Running example.** Consider the goal that a compiler correctly translates the Lustre equation  $y = a \text{ fby } b$ . Instead of treating this as one abstract claim, a safety case decomposes it into observable obligations: (G1) at the first cycle,  $y$  equals  $a_0$ ; (G2) at every later cycle  $n$ ,  $y$  equals  $b_{n-1}$ ; and (G3) the generated C state is updated consistently with the relevant clock. The argument states that the translation preserves the semantics of *fby* if these obligations hold. Corresponding evidence is then gathered to support each goal, such as concrete I/O test reports for G1 and G2, or formal proof transcripts for G3. This example also illustrates the practical role of the safety case: it records what has been checked, by which activity, under which bound, and which claims still lack evidence.

Safety cases currently enjoy widespread use across many fields. In the area of medical devices, manufacturers use safety cases to demonstrate safety compliance in device

manufacturing and usage processes [4], including infusion pumps [25] and cardiac pacemakers [18]. In autonomous driving, safety cases are extensively applied to argue the reliability of autonomous driving systems [16, 29]. Safety cases are also utilized in strict safety assessments of management processes in the aerospace sector and at nuclear facilities [20, 27, 31].

### 3 Trusta Extension and Top-Level Safety Case Construction

#### 3.1 LLM-Enhanced Trusta for Test Program Generation

Trusta is a desktop application developed by Chen et al. [11]. It is the first tool that uses large language models to support automatic safety case creation. We extended Trusta to support our research needs. In a safety case, each goal node defines a safety requirement, similar to a testing objective. We added a feature that can automatically generate Lustre test cases from a goal. The key idea is to use LLMs to produce test cases that match what the goal asks. For example, if the goal says “the compiler’s transformation of the `fby` operator is correct,” Trusta can generate Lustre programs using `fby` to test this.

We integrate the LLM via a modular architecture. As illustrated in Figure 2, this integration comprises four interconnected components designed to streamline the transition from user intent to executable test cases:

- User Interaction Module: This component serves as the interface for defining testing objectives. It allows users to select specific testing goals (via goal ID) and specify the desired quantity of test cases. It aggregates the input metadata and visualizes the generation results returned by the downstream modules.
- Prompt Engineering Module: It is responsible for context assembly. By utilizing the goal identifier and program context, it dynamically instantiates specific prompt templates to align with the user’s request.
- Middleware Service: Acting as an intermediary layer, this module handles data serialization and formatting to meet the API specifications. It also serves as a post-processor that parses the raw response from the LLM, extracting the valid test cases before returning them to the frontend.
- LLM Invocation Layer: This module manages the network communication with the LLM API. It executes the remote procedure calls, handles latency, and retrieves the raw inference results for the middleware.

In SCALC, the LLM-generated programs are used as supplementary inputs to the official test suites and are further used in the subsequent translation-rule review, differential testing, and bounded functional equivalence checking stages.

#### 3.2 Top-Level Safety Case Construction

To structure the assurance of Lustre compiler correctness, SCALC adopts a top-down decomposition strategy when constructing a safety case. As shown in Figure 3, the framework refines the overarching claim of compiler reliability into three verifiable sub-goals: (1) Lustre compiler translation rules are correct, (2) Lustre compiler translation

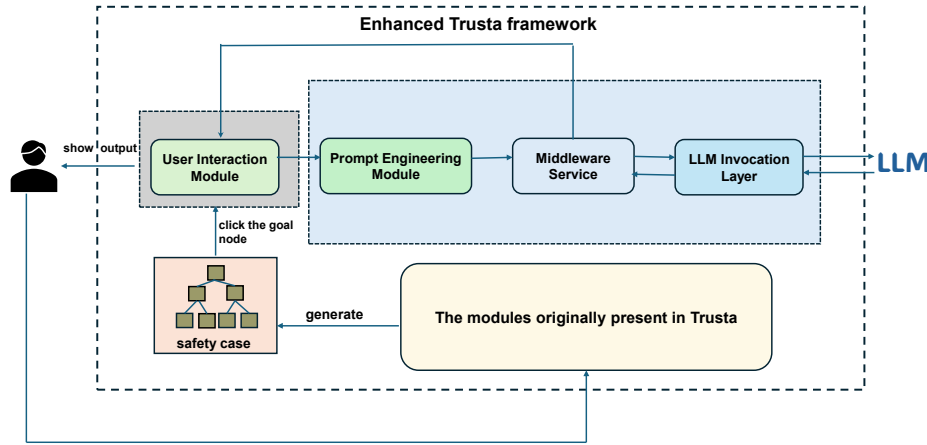


Fig. 2. Architecture of the LLM-integrated automated test generation pipeline in Trusta

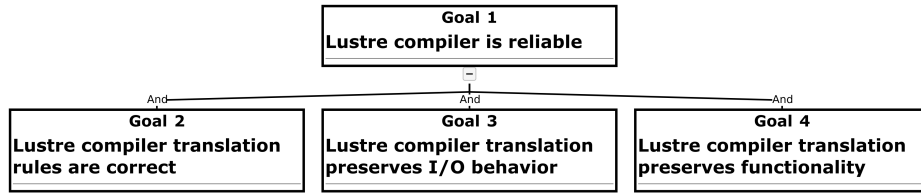


Fig. 3. Top-level safety case for Lustre compiler reliability

preserves I/O behavior, and (3) Lustre compiler translation preserves functionality. These sub-goals form the foundation of SCALC’s verification process.

The top-level safety case guides the subsequent analysis in a traceable manner. Each sub-goal determines a class of assurance activity: translation-rule correctness is examined by manual rule review, I/O-behavior preservation is assessed by differential testing, and functional preservation is checked by bounded equivalence analysis. The evidence produced by these activities is then linked back to the corresponding safety-case claims, forming an explicit argument from assurance goals to concrete validation results. In the following sections, these three sub-goals are elaborated in greater detail to illustrate the verification process within SCALC, and the L2C compiler is used as a practical case study to demonstrate the application of this framework.

## 4 Translation Rule Review

In this paper, translation rule review refers to a safety-case-guided manual examination of the compiler’s construct-level transformations. For each Lustre construct, we

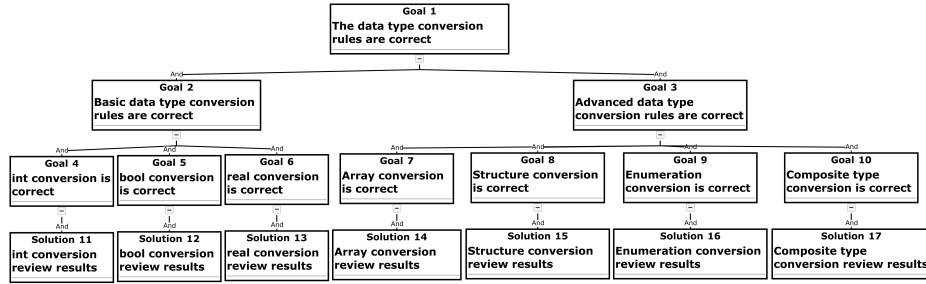


Fig. 4. Safety case for data type conversion rule review

check whether it is accepted according to the language specification and whether its intended semantics are preserved in the intermediate representations and translation results. We identify five key review areas covering the main transformations from dataflow to control flow. Data-type conversion and expression and operator conversion concern type mapping and computational semantics. Function and node-call conversion examines parameter passing, return values, and invocation behavior. Temporal-operator and array-iterator conversion and concurrency and synchronization conversion focus on Lustre-specific features, including temporal behavior, clocking, and synchronization. The extended Trusta generates test programs aligned with the corresponding safety-case goals to support the review. As illustrative examples, this section presents the reviews of data types, temporal operators, and array iterators; the identified defects are summarized in Section 7.

#### 4.1 Data Type Conversion Rule Review

Data type conversion rule checking is important for compiler validation, as wrong data type conversions may cause runtime errors including memory overflow, data loss, or type mismatches. Lustre’s basic data types include bool, int, and real, with advanced structures (array, enumeration, and structure) added in version V6. We build the safety case shown in Figure 4 for this review.

In the data type conversion rule review, we have found defects in L2C’s translation of structure types and enumeration types.

#### 4.2 Temporal Operators and Array Iterators Conversion Rule Review

The Lustre language can control dataflow through temporal operators and array iterators. The C language lacks similar mechanisms. Simulating their behavior in generated C code using loops and conditional statements therefore requires extremely precise compiler translation. We construct the safety case shown in Figure 5. In the temporal operators and array iterators conversion rule review, it turns out that the L2C compiler has a defect in translating the when operator.

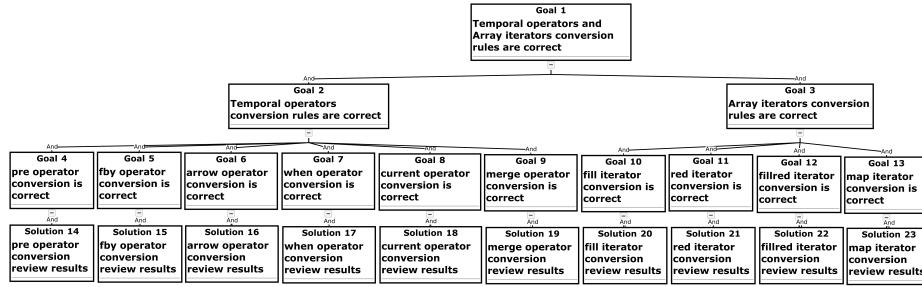


Fig. 5. Safety case for temporal operators and array iterators conversion rule review

## 5 Differential Testing for Lustre Compilers

SCALC incorporates differential testing between L2C and Vélus to identify behavioral inconsistencies. As an empirically effective bug-finding heuristic, differential testing leverages the principle that different compilers for the same language should produce semantically equivalent outputs for the same input. While correlated errors between compilers cannot be formally ruled out, the independent development and distinct translation pipelines of L2C and Vélus make identical failures unlikely in practice. Consequently, any disagreement in output provides strong evidence of a defect. The differential comparison also eliminates the need for a manual oracle and reduces the risk of subjective misjudgment. A disagreement is treated as a candidate defect rather than an immediate proof of an L2C bug; we manually inspect the Lustre source and both generated C programs to determine the root cause. In our experiments, the techniques employed include symbolic execution and fuzzing. Symbolic execution is a program analysis technique [21]. It can simulate the input process through symbolic variables and derive inputs that satisfy specific paths by analyzing the constraints in program execution paths. Fuzzing is an automated testing technique that triggers potential errors in the test subject by randomly generating or mutating test inputs [22, 28, 34]. Fuzzing can automatically generate a large number of different inputs to execute programs, thereby discovering program defects.

Figure 6 illustrates our testing strategy. Through differential testing, we reduce the question of whether L2C preserves a Lustre program’s I/O behavior to the question of whether the C programs generated by L2C and Vélus exhibit consistent I/O behavior. In our experiments, we use KLEE [8] to analyze C programs. By performing symbolic execution, KLEE generates test cases with high branch coverage. These test cases are then provided to AFL++ [15] as the input corpus, enabling AFL++ to mutate them and explore new execution paths, thus producing higher-quality test inputs.

Figure 7 shows the complete testing process in SCALC. Lustre source programs consist of programs automatically generated by Trusta and programs drawn from the official L2C and Vélus test sets. For each test case, we compile the Lustre source into C using both L2C and Vélus and add a main driver to each generated program.

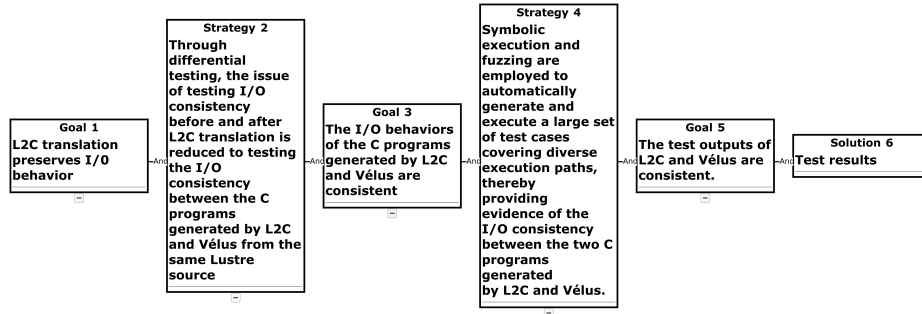


Fig. 6. Safety case for differential testing

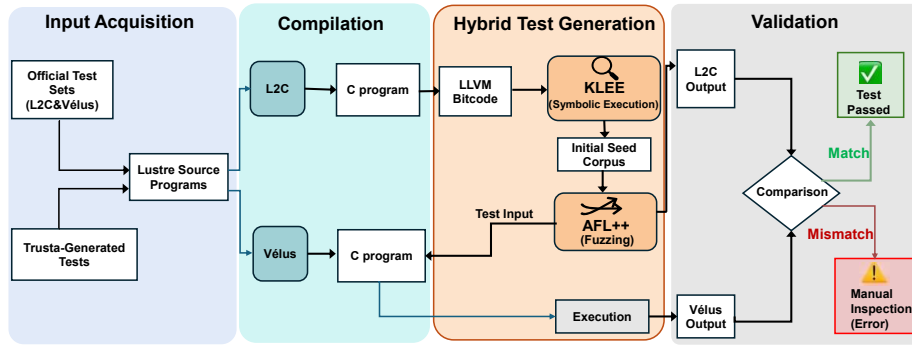


Fig. 7. Workflow for differential testing in SCALC

Then we convert the C code compiled by L2C into LLVM bitcode. KLEE automatically explores the execution paths of the program via dynamic symbolic execution, identifying which input variables can affect different execution paths, and generating test cases that cover as many branch structures as possible. We then use the obtained test cases to build an initial corpus for fuzzing. We input the test cases generated by KLEE as an initial corpus into AFL++. In this stage, AFL++ continuously mutates the initial corpus to generate a diverse suite of test inputs. These generated test inputs are systematically supplied to the C programs compiled by both L2C and Vélus to collect their respective execution outputs.

Finally, we compare these outputs to verify their consistency for each test case. If an inconsistency is detected, it indicates a potential defect, and we manually inspect the original Lustre program along with the two compiled C programs to investigate the error and determine its root cause.

This differential testing approach can be applied to any compiler that translates Lustre into C. We implement an automated process for test case generation and execution, eliminating the need for manual judgment of whether test cases should pass or fail; only

the consistency of outputs from C code needs to be compared. This effectively reduces the possibility of subjective misjudgment and reduces human intervention.

## 6 Bounded Functional Equivalence Checking

While testing identifies concrete defects, it cannot guarantee functional equivalence across all potential inputs. SCALC achieves a higher level of assurance by employing formal methods to exhaustively verify the generated C programs within a defined horizon. CBMC is a bounded model checking tool specifically designed for C and C++ code verification [12]. Additionally, CBMC can be used to check program equivalence [23, 26, 33]. We introduce a bounded functional equivalence checking approach based on CBMC and SMT solvers, which rigorously analyzes whether two programs produce identical outputs for all symbolic inputs within a specified cycle depth. This process involves three key steps: construction of a verification program with assertions, translation into SMT constraints, and constraint solving.

To check bounded functional equivalence between the C programs, we first merge them into a single verification program. This includes all function definitions and global variables from both programs, renaming as necessary to avoid naming conflicts. We then create a main function as the entry point. This function declares all input variables without setting specific values. We call the target functions from both programs using the same input parameters and add assertion statements like `assert(result_A == result_B)` to check if both functions give the same outputs with the same inputs. The inputs stay as symbolic variables instead of fixed values. This lets later stages explore all possible inputs through symbolic execution. Then, we use CBMC to convert the merged program to SMT constraint formulas. CBMC is run using an SMT output format and performs symbolic execution and model checking. It considers all possible execution paths and creates constraints that show when assertions hold or fail. CBMC converts the program logic into SMT constraints. The SMT file contains complete path constraints and the added assertions. The assertion is negated and becomes `(result_A != result_B)`. Finally, we use an SMT solver to solve the SMT constraints by finding input values that satisfy all constraints. If it returns “unsat,” the negated assertion cannot hold under the program constraints. This means both programs produce the same outputs for all valid inputs within the specified bound. If it returns “sat,” the programs are not equivalent and there are cases where the same inputs produce different outputs.

The safety case in Figure 8 shows the verification logic. In bounded functional equivalence checking, we encode the existence of an input sequence that causes different outputs as an SMT query. If the solver returns “unsat,” no such counterexample exists within the selected bound.

Since Lustre-generated C code operates as a reactive loop without unbounded recursion or dynamic memory allocation, the logic within each execution cycle is finite. By unwinding the loop to a fixed depth  $N$ , an “unsat” result provides bounded evidence that the two generated programs are equivalent for the first  $N$  cycles.

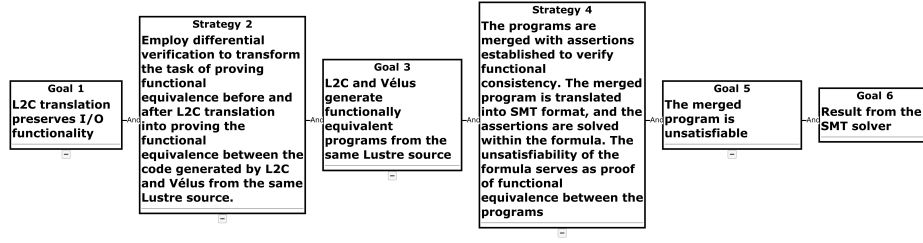


Fig. 8. Safety case for formal verification

## 7 Experimental Evaluation and Defect Analysis

### 7.1 Experimental Setup

To evaluate the effectiveness of SCALC, we applied it to the open-l2c-1.0 version of the L2C compiler. The evaluation focuses on whether SCALC can expose practical defects in a realistic, partially verified Lustre compiler. The test inputs were collected from two sources: Lustre programs generated from safety-case goals using the LLM-enhanced Trusta, and official test programs from L2C and Vélus. For temporal operators not supported by Vélus, we transformed them into semantically equivalent elementary constructs. Specifically, Trusta generated 20 test cases for each safety-case node. For each test case, the maximum execution time of KLEE and the fuzzing duration of AFL++ were set to 30 s and 2 s, respectively, and the generated programs were executed for 10 clock cycles. The bound  $N$  for bounded functional equivalence checking was set to 20.

All experiments evaluating the SCALC framework were conducted on a workstation equipped with an Intel Core Ultra 9 185H processor, 32 GB of RAM, and a 256 GB SSD, running a standard Linux distribution.

### 7.2 Defect Analysis in L2C

Applying the aforementioned multi-layered approach, SCALC identified seven distinct defects and two unsupported language features in L2C. The two unsupported features are the `assert` statement, used for specifying safety invariants, and array access through slicing. To localize the defects, we inspected the intermediate representations at each translation stage to determine the specific step where the erroneous behavior first appeared. Tables 2 and 3 summarize the seven defects by their characteristics, discovery methods, pipeline locations, and confirmation status. Overall, manual review mainly exposed a syntax incompatibility and semantic translation mismatches (D1–D3); differential testing revealed front-end acceptance discrepancies and a behavioral output discrepancy (D4–D6); and bounded functional equivalence checking isolated a temporal state-update error (D7).

**D1–D3: defects found by manual review.** Manual translation-rule review exposed defects in L2C’s handling of structures, enumeration types, and the `when` operator. D1 occurs during lexical and syntactic analysis: L2C does not support semicolons as

**Table 2.** Defects and their characteristics in the L2C compiler

| ID | Defect                                                            | Defect Characteristic            |
|----|-------------------------------------------------------------------|----------------------------------|
| D1 | Semicolon-separated structure elements unsupported                | Syntax incompatibility           |
| D2 | Enumeration return type translated to raw <code>int</code>        | Semantic translation mismatch    |
| D3 | Incorrect assignment translation of <code>when</code>             | Semantic translation mismatch    |
| D4 | Missing type checks for real <code>mod/div</code>                 | Front-end acceptance discrepancy |
| D5 | Multi-line block comments rejected                                | Front-end acceptance discrepancy |
| D6 | Incorrect precedence of <code>else</code> over temporal operators | Behavioral output discrepancy    |
| D7 | Incorrect state update for <code>current</code>                   | Temporal state-update error      |

**Table 3.** Discovery methods, locations, and status of identified defects

| ID | Detected by                             | Location                      | Status            |
|----|-----------------------------------------|-------------------------------|-------------------|
| D1 | Manual review                           | Lustre $\rightarrow$ AST      | Syntax difference |
| D2 | Manual review                           | AST $\rightarrow$ LustreV     | Fixed             |
| D3 | Manual review                           | LustreV $\rightarrow$ LustreS | Confirmed         |
| D4 | Differential testing                    | AST $\rightarrow$ LustreV     | Confirmed         |
| D5 | Differential testing                    | Lustre $\rightarrow$ AST      | Confirmed         |
| D6 | Differential testing                    | Lustre $\rightarrow$ AST      | Confirmed         |
| D7 | bounded functional equivalence checking | LustreS $\rightarrow$ LustreR | Confirmed         |

```

node test_mod(a:real; b:real) returns(c:real);
let
  c = a mod b;
tel

```

**Fig. 9.** Example of the mod operator

separators in structure definitions, which differs from the official Lustre syntax. D2 occurs in the AST-to-LustreV stage, where enumeration return types are incorrectly translated to raw `int`. D3 occurs in the LustreV to LustreS stage and causes incorrect temporal behavior during inactive clock cycles.

**D4–D6: defects found by differential testing.** Differential testing exposed three front-end and early-stage defects. D4 is a missing type-checking defect. As shown in Figure 9, L2C accepts a modulo operation over real operands, although Lustre only permits `mod` on integer operands. Similarly, Figure 10 shows an invalid use of the integer division operator `div` on real operands. L2C compiles both programs without reporting a source-level type error, causing the generated C code to deviate from Lustre’s operator semantics. D5 occurs during lexical analysis, where L2C rejects multi-line block comments. D6 occurs during parsing, where `else` has higher precedence than temporal operators such as `->` and `fbv`, leading to incorrect evaluation orders in conditional branches.

**D7: defect found by bounded functional equivalence checking.** D7 is the incorrect implementation of the `current` operator. For the test program in Figure 11, L2C translates it into the C function shown in Figure 12. The generated code updates `acg_fby1` with the current value of `i` in every cycle, even when `ck` is false. A correct translation should update this stored value only when `ck` is true.

```

node test_div(a:real; b:real) returns(c:real);
let
  c = a div b;
tel

```

**Fig. 10.** Example of the div operator

```

node test_current(ck:bool; i:int) returns(o:int);
let
  o = current(i when ck);
tel

```

**Fig. 11.** Example of the current operator

```

void test_current(inC_test_current *inC, outC_test_current *outC)
{
  if ((*inC).ck) {
    (*outC).o = (*inC).i;
  } else {
    if ((*outC).acg_init) {
      (*outC).o = 0;
    } else {
      (*outC).o = (*outC).acg_fby1;
    }
  }
  (*outC).acg_fby1 = (*inC).i;
  (*outC).acg_init = 0;
}

```

**Fig. 12.** L2C translation of the current operator

These results indicate that most defects reside in unverified early stages. In the Coq-verified LustreS to Clight part of the pipeline, SCALC found one defect. This suggests that L2C’s formal verification substantially strengthens the correctness of later translation steps, while semantic completeness remains a potential blind spot when the formalized semantics of an operator is incomplete or misaligned with the intended Lustre behavior.

### 7.3 Method Characteristics and Efficiency

In this subsection, we provide a comparative analysis of the three assurance activities to clarify their relative strengths and limitations. To quantify the execution cost of differential testing and bounded functional equivalence checking, each method was evaluated on 100 Lustre test programs covering scalar and temporal constructs supported end-to-end by both L2C and Vélus. Table 4 summarizes the execution efficiency of the two automated methods. One case denotes one Lustre source program. The KLEE and AFL++ stage times are end-to-end wall-clock measurements, including tool initialization and post-processing overhead. For differential testing, throughput denotes the number of concrete inputs per second replayed on the paired C programs generated by L2C and Vélus. All configurations follow Section 7.1.

**Table 4.** Execution efficiency of the automated assurance activities

| Method                       | Cases | Avg. time/case | Stage share                                                | Throughput     |
|------------------------------|-------|----------------|------------------------------------------------------------|----------------|
| Differential testing         | 100   | 40.88 s        | Prep 1.01%<br>KLEE 18.41%<br>AFL++ 25.65%<br>Replay 54.93% | 3,178 inputs/s |
| Bounded equivalence checking | 100   | 4.92 s         | Model prep 0.17%<br>CBMC 1.15%<br>SMT 98.68%               | N/A            |

**Manual translation-rule review.** Manual translation-rule review primarily exposed a syntax incompatibility (D1) and semantic translation mismatches (D2–D3). It can compare the intended semantics with the intermediate representation and the translation results, even when these do not directly produce an observable input/output discrepancy. Guided by safety-case goals, SCALC focuses manual review on semantically critical transformations to reduce review effort; however, it remains less scalable than automated activities, depends heavily on reviewer expertise, and cannot guarantee exhaustive coverage.

**Differential testing.** Differential testing provides concrete, executable evidence and can expose both compiler acceptance differences at the front end (D4 and D5) and output differences after both toolchains generate executable C code (D6). Although its average end-to-end time per case (40.88 s) is longer than that of bounded equivalence checking, differential testing achieves a replay throughput of 3,178 inputs/s because the compiled C programs can efficiently process large batches of concrete inputs.

**Bounded functional equivalence checking.** Bounded functional equivalence checking complements testing by searching all symbolic input sequences within the selected bound, making it highly effective at isolating deep logical errors in temporal state updates (D7). As shown in Table 4, bounded equivalence checking averaged 4.92 s per case in our experiments, lower than differential testing, partly because it directly analyzes a bounded symbolic model without the additional cost of test-input generation and concrete replay. Its limitations include the rapidly increasing solver cost as the verification bound grows; consequently, any conclusions are restricted to the selected execution horizon. Thus, bounded checking is powerful for supported workloads but cannot replace front-end testing or real-toolchain execution.

## 8 Related Work

Research on compiler assurance for the Lustre language has primarily focused on formal verification. Compilers such as L2C [32] and Vélus [6] employ formal methods to prove translation correctness, yet they face distinct limitations: L2C lacks a comprehensive pipeline-wide proof, while Vélus provides end-to-end verification at the cost of supporting only a restricted subset of high-level language features. Furthermore, formal verification is fundamentally constrained by the accuracy of its underlying specifications. For instance, SCALC identified a defect within a formally verified L2C module,

stemming from a semantic misalignment in the operator definition. To mitigate the complexity of full verification, Brun et al. [7] proposed translation validation for the LustreC compiler, but this approach remains restricted to a core language subset and lacks support for complex data types like arrays. Beyond compiler correctness, tools such as Kind 2 [10] and NKind [35] verify Lustre application logic. These tools verify properties of the Lustre source model but do not address whether subsequent compilation preserves those properties.

SCALC bridges this gap with a multi-layered assurance framework that integrates safety-case-guided review, hybrid differential testing, and bounded functional equivalence checking. SCALC offers a systematic solution for identifying potential translation defects and ensuring the reliability of Lustre compiler pipelines in safety-critical domains.

## 9 Conclusion

We presented SCALC, a safety-case-guided assurance framework for Lustre compilers. SCALC combines manual translation-rule review, hybrid differential testing, and bounded functional equivalence checking within a structured safety-case argument, and uses LLM-assisted generation as supplementary support for producing safety-case-aligned Lustre tests. Applying SCALC to the open-source L2C compiler, we identified seven defects and two unsupported language features. SCALC provides a reusable and systematic model for assuring the reliability of Lustre compilers in embedded safety-critical domains.

**Acknowledgments.** The work was supported by the National Natural Science Foundation of China under Grant No. 62472175 and the Fundamental Research Funds for the Central Universities.

**Disclosure of Interests.** The authors have no competing interests to declare that are relevant to the content of this article.

## References

1. Barras, B., Boutin, S., Cornes, C., Courant, J., Coscoy, Y., Delahaye, D., et al.: The Coq proof assistant reference manual. Tech. Rep. Version 6.3.1, INRIA (1999), <https://pauillac.inria.fr/~herbelin/tmp/coq/V6/V6.3/doc/www/cover.html>
2. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer (2004). <https://doi.org/10.1007/978-3-662-07964-5>
3. Bishop, P., Bloomfield, R.: A methodology for safety case development. In: Redmill, F., Anderson, T. (eds.) Industrial Perspectives of Safety-critical Systems: Proceedings of the Sixth Safety-critical Systems Symposium. pp. 194–203. Springer (1998). [https://doi.org/10.1007/978-1-4471-1534-2\\_14](https://doi.org/10.1007/978-1-4471-1534-2_14)
4. Bloomfield, R., Chozos, N., Cleland, G., Adelard LLP: Safety case use within the medical devices industry. In: Supplements to: Using Safety Cases in Industry and Healthcare, pp. 75–91. The Health Foundation, London (2012)

5. Bloomfield, R.E., Bishop, P.G.: Safety and assurance cases: Past, present and possible future – an Adelard perspective. In: Dale, C., Anderson, T. (eds.) *Making Systems Safer: Proceedings of the Eighteenth Safety-Critical Systems Symposium*. pp. 51–67. Springer (2010). [https://doi.org/10.1007/978-1-84996-086-1\\_4](https://doi.org/10.1007/978-1-84996-086-1_4)
6. Bourke, T., Brun, L., Dagand, P.É., Leroy, X., Pouzet, M., Rieg, L.: A formally verified compiler for Lustre. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. pp. 586–601. ACM (2017). <https://doi.org/10.1145/3062341.3062358>
7. Brun, L., Garion, C., Garoche, P.L., Thirioux, X.: Equation-directed axiomatization of Lustre semantics to enable optimized code validation. *ACM Transactions on Embedded Computing Systems* **22**(5s), 151:1–151:24 (2023). <https://doi.org/10.1145/3609393>
8. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*. pp. 209–224. USENIX Association (2008), [http://www.usenix.org/events/osdi08/tech/full\\_papers/cadar/cadar.pdf](http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf)
9. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.A.: Lustre: A declarative language for programming synchronous systems. In: *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages*. pp. 178–188. ACM Press (1987). <https://doi.org/10.1145/41625.41641>
10. Champion, A., Mebsout, A., Stickse, C., Tinelli, C.: The Kind 2 model checker. In: Chaudhuri, S., Farzan, A. (eds.) *Computer Aided Verification: Proceedings of the 28th International Conference, Part II. Lecture Notes in Computer Science*, vol. 9780, pp. 510–517. Springer (2016). [https://doi.org/10.1007/978-3-319-41540-6\\_29](https://doi.org/10.1007/978-3-319-41540-6_29)
11. Chen, Z., Deng, Y., Du, W.: Trusta: Reasoning about assurance cases with formal methods and large language models. *Science of Computer Programming* **244**, 103288 (2025). <https://doi.org/10.1016/J.SCICO.2025.103288>
12. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science*, vol. 2988, pp. 168–176. Springer (2004). [https://doi.org/10.1007/978-3-540-24730-2\\_15](https://doi.org/10.1007/978-3-540-24730-2_15)
13. Colaço, J.L., Pagano, B., Pouzet, M.: SCADE 6: A formal language for embedded critical software development (invited paper). In: *Proceedings of the 11th International Symposium on Theoretical Aspects of Software Engineering*. pp. 1–11. IEEE Computer Society (2017). <https://doi.org/10.1109/TASE.2017.8285623>
14. Deng, Y., Chen, Z., Du, W., Mao, B., Liang, Z., Lin, Q., et al.: Trustworthiness derivation tree: A model of evidence-based software trustworthiness. In: *Proceedings of the 2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion*. pp. 487–493. IEEE (2021). <https://doi.org/10.1109/QRS-C55045.2021.00077>
15. Fioraldi, A., Maier, D., Eißfeldt, H., Heuse, M.: AFL++: Combining incremental steps of fuzzing research. In: *Proceedings of the 14th USENIX Workshop on Offensive Technologies*. USENIX Association (2020), <https://www.usenix.org/conference/woot20/presentation/fioraldi>
16. Griessnig, G., Schnellbach, A.: Development of the 2nd edition of the ISO 26262. In: *Proceedings of the 24th European Conference on Systems, Software and Services Process Improvement. Communications in Computer and Information Science*, vol. 748, pp. 535–546. Springer (2017). [https://doi.org/10.1007/978-3-319-64218-5\\_44](https://doi.org/10.1007/978-3-319-64218-5_44)
17. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE* **79**(9), 1305–1320 (1991). <https://doi.org/10.1109/5.97300>

18. Jee, E., Lee, I., Sokolsky, O.: Assurance cases in model-driven development of the pace-maker software. In: *Leveraging Applications of Formal Methods, Verification, and Validation. Lecture Notes in Computer Science*, vol. 6416, pp. 343–356. Springer (2010). [https://doi.org/10.1007/978-3-642-16561-0\\_33](https://doi.org/10.1007/978-3-642-16561-0_33)
19. Kang, Y.X., Gan, Y.K., Wang, S.Y.: Comparison of two trustworthy compilers Vélus and L2C for synchronous languages (in Chinese). *Journal of Software* **30**(7), 2003–2017 (2019). <https://doi.org/10.13328/j.cnki.jos.005755>
20. Kelly, T., Bate, I., McDermid, J., Burns, A.: Building a preliminary safety case: An example from aerospace. In: *Proceedings of the Australian Workshop on Industrial Experience with Safety Critical Systems and Software*. pp. 1–10. Australian Computer Society (1997)
21. King, J.C.: Symbolic execution and program testing. *Communications of the ACM* **19**(7), 385–394 (1976). <https://doi.org/10.1145/360248.360252>
22. Klees, G., Ruef, A., Cooper, B., Wei, S., Hicks, M.: Evaluating fuzz testing. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. pp. 2123–2138. ACM (2018). <https://doi.org/10.1145/3243734.3243804>
23. Klemmer, L., Große, D.: EPEX: processor verification by equivalent program execution. In: *Proceedings of the Great Lakes Symposium on VLSI*. pp. 33–38. ACM (2021). <https://doi.org/10.1145/3453688.3461497>
24. Lan, L., Ma, Q., Hou, R.B., Jiang, W., Liu, M.X., Yang, F., et al.: Formally verified a front-end of the trusted code generator for safety I&C software of NPPs. In: *Nuclear Power Plants: Innovative Technologies for Instrumentation and Control Systems. Lecture Notes in Electrical Engineering*, vol. 779, pp. 260–272. Springer (2021). [https://doi.org/10.1007/978-981-16-3456-7\\_26](https://doi.org/10.1007/978-981-16-3456-7_26)
25. Larson, B.R., Hatcliff, J., Chalin, P.: Open source patient-controlled analgesic pump requirements documentation. In: *Proceedings of the 5th International Workshop on Software Engineering in Health Care*. pp. 28–34. IEEE Computer Society (2013). <https://doi.org/10.1109/SEHC.2013.6602474>
26. Lee, D.A., Yoo, J., Lee, J.S.: Equivalence checking between function block diagrams and C programs using HW-CBMC. In: *Proceedings of the 30th International Conference on Computer Safety, Reliability, and Security. Lecture Notes in Computer Science*, vol. 6894, pp. 397–408. Springer (2011). [https://doi.org/10.1007/978-3-642-24270-0\\_29](https://doi.org/10.1007/978-3-642-24270-0_29)
27. Leveson, N.: The use of safety cases in certification and regulation. Tech. Rep. ESD-WP-2011-13, Massachusetts Institute of Technology, Engineering Systems Division (2011), <https://hdl.handle.net/1721.1/102833>
28. Liang, H., Pei, X., Jia, X., Shen, W., Zhang, J.: Fuzzing: State of the art. *IEEE Transactions on Reliability* **67**(3), 1199–1218 (2018). <https://doi.org/10.1109/TR.2018.2834476>
29. Palin, R., Habli, I.: Assurance of automotive safety - A safety case approach. In: *Proceedings of the 29th International Conference on Computer Safety, Reliability, and Security. Lecture Notes in Computer Science*, vol. 6351, pp. 82–96. Springer (2010). [https://doi.org/10.1007/978-3-642-15651-9\\_7](https://doi.org/10.1007/978-3-642-15651-9_7)
30. Perez, I., Mavridou, A., Pressburger, T., Goodloe, A., Giannakopoulou, D.: Automated translation of natural language requirements to runtime monitors. In: *Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science*, vol. 13243, pp. 387–395. Springer (2022). [https://doi.org/10.1007/978-3-030-99524-9\\_21](https://doi.org/10.1007/978-3-030-99524-9_21)
31. Rushby, J., Xu, X., Rangarajan, M., Weaver, T.L.: Understanding and evaluating assurance cases. Tech. Rep. NASA/CR-2015-218802, NASA Langley Research Center (2015), <https://ntrs.nasa.gov/citations/20160000772>
32. Shang, S., Gan, Y.K., Shi, G., Wang, S.Y., Dong, Y.: Key translations of the trustworthy compiler L2C and its design and implementation (in Chinese). *Journal of Software* **28**(5), 1233–1246 (2017). <https://doi.org/10.13328/j.cnki.jos.005213>

33. Staats, M., Heimdahl, M.P.E.: Partial translation verification for untrusted code-generators. In: Proceedings of the 10th International Conference on Formal Engineering Methods. Lecture Notes in Computer Science, vol. 5256, pp. 226–237. Springer (2008). [https://doi.org/10.1007/978-3-540-88194-0\\_15](https://doi.org/10.1007/978-3-540-88194-0_15)
34. Takanen, A., DeMott, J.D., Miller, C., Kettunen, A.: Fuzzing for software security testing and quality assurance. Artech House, 2nd edn. (2018)
35. Wei, J., Li, Q.: NKind: a model checker for liveness property verification on Lustre programs. In: Peng, R., Pantoja, C.E., Kamthan, P. (eds.) Proceedings of the 34th International Conference on Software Engineering and Knowledge Engineering. pp. 351–356. KSI Research Inc. (2022). <https://doi.org/10.18293/SEKE2022-089>
36. Winandy, I., Dion, A., Garoche, P.L., Manni, F.: A reactive system-specific compilation chain from synchronous dataflow models to FPGA netlist. In: Proceedings of the 10th IEEE International Conference on Space Mission Challenges for Information Technology. pp. 11–21. IEEE (2024). <https://doi.org/10.1109/SMC-IT61443.2024.00009>