Master's Thesis

Circular assume-guarantee reasoning of synchronous systems using Kind 2 and Z3Py

Ngo Tien Duc

Supervisor Daisuke Ishii

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
(Information Science)

August, 2023

Abstract

Background. SMT-based model checking (MC) is a technique for verifying the correctness of a system using satisfiability modulo theories (SMT) solvers. SMT solvers are a tool for checking the satisfiability of first-order predicate logic formulas. In SMT-based MC, a system model (described as a state transition system) and a temporal property are encoded into a set of logic formulas; then, an SMT solver is applied to verify whether the system satisfies the property. As a target system becomes large and complex, so do the encoded logic formulas, leading to an increased execution time of the verification.

Kind 2 is an SMT-based MC tool that is designed for Lustre programs. Lustre is a synchronous data-flow programming language that is often used in safety-critical system development. Kind 2 deals with safety properties on a program P. The input is an annotated program $\{A\}P\{G\}$ (called triple) where A and G are sets of properties (an assume-guarantee contract). Kind 2 verifies the correctness of the triple; namely, for any input signal always satisfying A, the output signal always satisfies G. Kind 2 implements standard SMT-based MC methods such as the BMC and k-induction methods. Kind 2 supports compositional verification to improve the efficiency of the MC process based on the node-wise component structure of Lustre programs. In a compositional MC process, Kind 2 considers a set of annotated nodes $\{\{A_1\}N_1\{G_1\}, \ldots, \{A_n\}N_n\{G_n\}\}$ and verifies that each of them is correct. In the MC of parent nodes, the descriptions of child nodes N_i are abstracted with their contracts A_i and G_i to make the process efficient.

Kind 2 has limitations in handling practical Lustre programs when a given property depends on lengthy behaviors or when a node description contains circular referencing of signals. Lengthy behaviors result in large logic formulas in the MC process and the compositional process can be effective to analyze them. Circular referencing of signals occurs when two nodes N and N' depend on each other. For instance, the output of N provides the input for N' and *vice versa*. When verifying a program containing such a circular pair of nodes, the compositional MC function of Kind 2 does not work, resulting in a spurious counterexample.

Objectives. This thesis proposes a new compositional MC method for Lustre programs that can handle circular programs. The research consists of the following activities: 1) We study the relationship between Lustre programs and the theory of synchronous modules. Lustre nodes can be translated to

synchronous modules so that the existing compositional reasoning methods can be applied. For instance, this will enable to apply a reasoning rule for circular composite modules. 2) We study the automation of deductive reasoning based on the composition rules. Using the Z3 SMT solver with compositional reasoning rules encoded as universally quantified formulas, it is possible to automate the proposed method. We implement a tool that translates Lustre nodes to synchronous modules and then performs compositional reasoning. 3) We conduct experiments based on examples to evaluate the validity and performance of the proposed method.

Originality. Circular cases are only supported partially in Kind 2, so we study how to perform compositional verification more completely. Our method is original because it is different from the compositional verification function of Kind 2, as we regard Lustre node instances as synchronous modules and exploit the dedicated deduction rules for circular reasoning.

Significance. Verifying the safety of practical synchronous systems is important to prevent accidents and ensure correct operation, especially in safety-critical systems. For example, Kind 2 and our method can be used to verify a control system for a transport-class aircraft.

Practical synchronous systems are described as a large and complex set of Lustre nodes, possibly involving circular referencing. Our compositional MC method is significant because the compositional approach is an effective way to improve the scalability of verification tools.

Methodology. This research contains a survey on SMT solvers, synchronous modules, model checking methods, and the Kind 2 tool.

We design a method whose input is an annotated Lustre program and output is the correctness of the input. The basic steps of the proposed method are: 1) *Translation from Lustre nodes into synchronous modules.* We convert Lustre node instances NI_i and properties AI_i and GI_i into synchronous modules $M(NI_i)$, $M(AI_i)$ and $M(GI_i)$, respectively. The we interpret triples $\{AI_i\}\ NI_i\ \{GI_i\}$ into implementation relations on the modules $M(NI_i)\ ||\ M(AI_i) \models M(GI_i)$. 2) *Automatic construction of a proof tree based on the deduction rules.* We encode the definition of the compositional reasoning rules, the declarations of modules, and the implementation relations among modules into a set of logic formulas. Then, we run an SMT solving process to search for a proof tree. Kind 2 is used to check the correctness of leaf nodes. We discuss the soundness of the method. It is followed from the correctness of the translation between Lustre nodes and synchronous modules and the soundness of the deduction rules.

We have implemented the method as a Python script using Kind 2 and Z3Py. The tool consists of: 1) *A translator module.* Given node instances and annotated properties, the module generates an intermediate representation of synchronous modules. 2) *A printer module* that prints intermediate data, such as a monolithic form of the input program and separated Lustre node instances. 3) *A validator module* that performs the proof tree search using Z3Py.

Evaluation. We conducted an experiment to compare the execution time of a monolithic verification process of Kind 2 and a compositional verification process of the proposed method.

We prepared an example of an integrator delayed by two nested counters. The execution time increased because the number of execution steps required to be analyzed in the verification increased as the values of the two parameters increased. In the experiment, we observed correlations between the parameter values, the number of analyzed steps, and the execution time. Along with parameter values, execution time increased rapidly (up to 16 times for one increase) for the monolithic process, while the compositional process took much less time (up to a 150-fold improvement). The reason is that the number of analyzed steps in the monolithic verification was affected by the delay of the two counters, while compositional verification could verify each node separately without taking into account the delay.

We conducted another experiment to evaluate whether our implementation verifies the Lustre programs correctly and efficiently. We prepared two circular examples that cannot be handled by the compositional verification function of Kind 2: 1) A simple parallel composition of synchronous modules. 2) A more complex example containing two digital filters. We confirmed that our implementation verified the two examples correctly. Besides, the process for the second example was inefficient due to the large search space among many possible deductions in the compositional reasoning.

Conclusion. In the research, we studied how to apply deduction rules for circular reasoning so that we can perform compositional verification on Lustre programs with a different approach from the Kind 2 tool.

Keywords: SMT-based model checking, Lustre programming language, synchronous systems, assume-guarantee reasoning, compositional verification

Acknowledgements

First, I would like to express my sincere gratitude to my supervisor, Associate Professor Daisuke Ishii. Although the time was limited and I was totally new to this research area, he was always patient and gave me guidance step by step. I have learned a lot from my supervisor, not only knowledge but also academic skills. I would not be able to complete my master's program on time without his enthusiastic instructions.

I want to say thank you to all the Professors and lecturers at JAIST for teaching me and giving me advice. I am also grateful to my teachers at PTIT, my university in Vietnam. They provided me with a lot of assistance so that I could come to Japan to study and have unforgettable experiences.

I would like to send my thanks to my friends at JAIST for their encouragement. My daily life will be boring without them. A special thank you to members from Aoki-lab and Ogata-lab for giving me helpful instructions on my first days here.

Finally, thank you, my family and friends from Vietnam, for always supporting me on my way.

Ngo Tien Duc.

Contents

This thesis was prepared according to the curriculum for the Collaborative Education Program organized by **Japan Advanced Institute of Science and Technology** and **Posts and Telecommunications Institute of Technology**.

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

SMT-based model checking (MC) (Section 2.2) is a technique for verifying the correctness of a system or program using Satisfiability Modulo Theories (SMT) solvers (Section 2.1). SMT solvers are a type of logic solver that can reason about first-order logic formulas with constraints from various theories, such as arithmetic, bit-vectors, and arrays. In SMT-based MC, the system is modeled as a state transition system. SMT solvers are then applied to verify whether a model satisfies the property. SMT-based MC is important for safety-critical systems because it ensures the correct operation of the systems. As a target system becomes larger and more complex, so do the encoded logic formulas, leading to an increased execution time for the verification.

Verifying the safety of those systems is important to prevent accidents and ensure correct operation, especially safety-critical systems. For example, a software error in the Therac-25, a machine that was used to treat cancer patients, caused it to deliver massive overdoses of radiation to several patients, resulting in their deaths [6]. The accident showed that any software errors could cause serious consequences. After the accident, there has been a growth in the effort to verify safety-critical systems.

Kind 2 (Section 2.5) is an SMT-based MC tool that is designed for Lustre programs. Lustre (Section 2.3) is a synchronous data-flow programming language that is often used in safety-critical system development. Kind 2 deals with safety properties on a program P. The input is an annotated program

$\{A\}P\{G\}$ (called triple) where A and G are sets of properties (an assume-guarantee contract). Kind 2 verifies the correctness of the triple; namely, for any input signal always satisfying A, the output signal always satisfies G. Kind 2 implements standard SMT-based MC methods such as the BMC and k-induction methods. Kind 2 supports compositional verification to improve the efficiency of the MC process based on the node-wise component structure of Lustre programs. In a compositional MC process, Kind 2 considers a set of annotated nodes $\{\{A_1\}N_1\{G_1\}, \ldots, \{A_n\}N_n\{G_n\}\}$ and verifies that each of them is correct. In the MC of parent nodes, the descriptions of child nodes N_i are abstracted with their contracts A_i and G_i to make the process efficient.

Kind 2 has limitations in handling practical Lustre programs when a given property depends on lengthy behaviors or when a node description contains circular referencing of signals. Lengthy behaviors result in large logic formulas in the MC process and the compositional process can be effective to analyze them. Circular referencing of signals occurs when two nodes N and N' depend on each other. For instance, the output of N provides the input for N' and *vice versa*. During compositional reasoning, it is also often necessary to assume the correctness of N to verify N' and *vice versa* [7]. When verifying a program containing such a circular pair of nodes, the compositional MC does not work, resulting in a spurious counterexample.

1.2 Objective

In this thesis, we propose a new compositional MC method for Lustre programs that can handle circular programs and implement the method as a tool using Kind 2 and Z3Py. Our research consists of the following activities:

- We study the relationship between Lustre programs and the theory of synchronous modules. Lustre nodes can be translated to synchronous modules so that the existing compositional reasoning methods can be applied. For instance, this will enable to apply a reasoning rule for circular composite modules.

- We study the automation of deductive reasoning based on the composition rules. Using the Z3 SMT solver with compositional reasoning rules encoded as universally quantified formulas, it is possible to automate the proposed method. We implement a tool that translates Lustre

nodes to synchronous modules and then performs compositional reasoning.

- We conduct experiments based on examples to evaluate the validity and performance of the proposed method.

The main contributions of this research include a compositional verification method for circular Lustre programs, a tool implementation of the method, and the experimental results.

1.3 Summary of the research

Originality

Circular cases are only supported partially in Kind 2, so we study how to support them more completely. Our method is original because it is different from the compositional verification function of Kind 2, as we regard Lustre node instances as synchronous modules and exploit the dedicated deduction rules for circular reasoning.

Significance

Verifying the safety of practical synchronous systems is important to prevent accidents and ensure correct operation, especially in safety-critical systems. For example, Kind 2 and our method can be used to verify a control system for a transport-class aircraft.

We investigate a compositional safety MC method for Lustre programs based on the Kind 2 tool and Z3Py. Our method is significant because the compositional approach is an effective way to improve the scalability of verification tools, and synchronous systems may be described as a large and complex set of Lustre nodes. Also, circular referencing often occurs in practical systems.

Methodology

We design our method whose input is an annotated Lustre program and output is the correctness of the input. The basic steps of the proposed method are:

1. Translation from Lustre nodes into synchronous modules: We convert Lustre node instances NI_i and properties AI_i and GI_i into synchronous modules $M(NI_i)$, $M(AI_i)$ and $M(GI_i)$, respectively. The we interpret triples $\{AI_i\}\ NI_i\ \{GI_i\}$ into implementation relations on the modules $M(NI_i)\ ||\ M(AI_i) \models M(GI_i)$.

2. Automatic construction of a proof tree based on the deduction rules: We encode the definition of the compositional reasoning rules, the declarations of modules, and the implementation relations among modules into a set of logic formulas. Then, we run an SMT solving process to search for a proof tree. Kind 2 is used to check the correctness of leaf nodes.

We implemented a tool as a Python script using Z3Py and Kind 2, which consists of a translator, a printer, and a validator module. Then we conducted experiments to evaluate the efficiency of the proposed method and confirmed that our implementation verified the examples correctly.

1.4 Organization of the thesis

The remainder of this thesis is organized as follows:

- **Chapter 2: Preliminaries**
  This chapter introduces background knowledge. We introduce SMT solvers, Z3Py, a Python API for the Z3 theorem prover, and SMT-based MC. We also illustrate the Lustre language, the Kind 2 tool with the assume-guarantee contract, and compositional verification.

- **Chapter 3: Formalization**
  This chapter shows the formalization of the proposed method. We describe the basic synchronous model, including definitions of synchronous modules, implementation relation, compatibility, and parallel composition operation. Then we describe the interpretation from Lustre nodes to synchronous modules.

- **Chapter 4: Proposed method**
  In this chapter, we illustrate our proposed method for checking the correctness of Lustre programs annotated with assume-guarantee contracts. The chapter includes the problem statement, process of the

method, compositional programs interpretation, deduction rules, and some lemmas we used for reasoning.

- **Chapter 5: Experiments**
  This chapter describes the implementation of our proposed method using Kind 2 and Z3Py. Examples and our experiments to evaluate the method are also presented.

- **Chapter 6: Related work**
  This chapter mentions some studies related to our research.

- **Chapter 7: Conclusion**
  The final chapter summarizes the main contents of the thesis, our proposed method along with some limitations and future work based on the research.

Chapter 2

Preliminaries

This chapter describes background knowledge. We introduce SMT solvers and Z3Py (a Python API for the Z3 theorem prover) in Section 2.1, SMT-based model checking in Section 2.2, the Lustre language in Section 2.3, assume-guarantee contracts involving LTL properties in Section 2.4, and the Kind 2 tool in Section 2.5.

2.1 SMT solvers and Z3Py

2.1.1 SMT solvers

Satisfiability modulo theories (SMT) [13] generalizes Boolean satisfiability by adding equality reasoning, integer and real arithmetic, fixed-size bit-vectors, arrays, quantifiers, and other useful first-order theories. *SMT solvers* are automated tools that can check the satisfiability of first-order predicate logic formulas. The satisfiability of a given formula refers to the existence of a valid assignment of variables that makes the formula true.

SMT solvers [13] can handle complex formulas and explore large search spaces. They have practical applications in extended static checking, predicate abstraction, test case generation, etc.

2.1.2 Z3Py

Z3Py[1] is a Python API for Z3. *Z3* [17] is a powerful SMT solver developed by Microsoft Research. It is widely used for automated reasoning, formal verification, program analysis, and constraint solving.

Z3Py provides a user-friendly way to interact with the Z3 solver through the Python programming language. Users can construct logical formulas and constraints in Python, and then use the Z3 solver to check the satisfiability of these formulas. Some advantages of Z3Py:

- *Python Integration.* Z3Py allows users to leverage the full capabilities of the Z3 solver using Python, a popular and easy-to-use programming language. This integration makes it more accessible for users who are familiar with Python.
- *Syntax.* Z3Py can make the construction of logical formulas more natural and intuitive since it provides a high-level syntax that aligns with the syntax of Python.
- *SMT Solver Functionality.* Z3Py supports a wide range of logical theories and constraints, such as arithmetic, bit-vectors, arrays, quantifiers, etc.
- *Performance.* Z3 is able to handle large and complex formulas, which is suitable for formal verification.
- *Customizability.* Z3Py allows users to customize the behavior of the solver using several options and parameters.

Example 1. We implemented bounded model checking (BMC) method [14] in Z3Py to verify the satisfiability of an integrator (Fig. 2.1).

We have an integrator whose input is a stream of real numbers ranging from 0 to 1. Output stream values are computed as follows

$$out_i = \begin{cases} in_i & \text{if } i = 1, \\ in_i + 0.9 * out_{i-1} & \text{otherwise.} \end{cases}$$

where s is the step number, in_i and out_i are values of input and output at step s. In this program, we want to check that the output value will be greater than 9.9 in a step. The result is "`sat in step 43`," which means the output value in step 43 of the stream can be over 9.9. □

[1] `https://z3prover.github.io/api/html/namespacez3py.html`

```
def bmc(i, y0):
  x1, y1 = Consts("x%d y%d" % (i,i), RealSort())
  s.add(And(x1 <= 1.0, x1 >= 0.0, y1 == x1 + 0.9 * y0))
  s.push()
  if s.check(y1 > 9.9) == sat:
    print("sat in step", i)
    m = s.model()
    for j in range (0, i+1):
      ys = Real("y%d" % j)
      print("step %d: output = %r" % (j,m[ys]))
    return
  s.pop()
  bmc(i+1, y1)

y0 = Real("y0")
set_option(rational_to_decimal=True)
s = Solver()
s.add(And(y0 <= 1.0, y0 >= 0.0))
s.push()
if s.check(y0 > 9.9) == sat:
  print("sat in step 0")
  print(s.model())
s.pop()
bmc(1, y0)
```

Figure 2.1: A Python script for the BMC using Z3Py.

2.2 SMT-based model checking

SMT-based model checking [1] is a formal verification technique used to verify the correctness of software or hardware systems, which is based on SMT solvers and the model checking (MC) methods. *MC* [2] is a formal method to explore all possible states of a system and verify whether a given property holds. It involves constructing a finite-state model of the system and exploring its state space to check the specified properties.

In SMT-based MC, a model is described as a state transition system, and a temporal property is encoded into a set of logic formulas. An SMT solver is then applied to verify whether a model satisfies the property.

SMT-based MC has several advantages, such as the ability to handle a wide range of theories and complex logical formulas or provide more efficient solutions than traditional exhaustive MC. The automation provided by SMT solvers is also helpful for improving the velocity of the verification process.

However, SMT-based MC also has limitations, such as the problem of state space explosion [5], which often occurs when the number of states in the system's model becomes too large to handle efficiently. Several research activities on improving SMT solvers have been conducted for finding solutions and developing new techniques to overcome these challenges and make SMT-based MC more effective and scalable for verifying complex systems.

2.3 The Lustre programming language

Lustre [3, 9] is a synchronous data-flow programming language that is suitable for describing reactive systems. A reactive system is composed of components that can react to changes in their environment (e.g., parent component or the physical environment) in real-time. The description of a Lustre program can be separated into *nodes*. Inputs and outputs of a node are streams, i.e., a sequence of values of the same type. The execution is divided into discrete time steps, and all variables in the system are updated simultaneously and consistently at every step. At the n-th execution step (or cycle) of the program, all the involved streams take their n-th value. The synchronous and data-flow model of Lustre ensures predictability and determinism. These characteristics make Lustre suitable for developing safety-critical systems, where accuracy and real-time responsiveness are very important.

Example 2. In Lustre, a node can be defined as:

```
node Sample (in: int) returns (out: int)
var c: int;
let
  c = 2 ;
  out = in -> (in + c * pre out) ;
tel
```

A Lustre node begins with its declaration, which includes the keyword `node` and the name of the node. This name must be unique within a program. Input variables are specified after the node's name. Output variables are specified after the keyword `returns`. *Auxiliary variables* (or local variables) can be declared by keyword `var`. These variables are used for naming

Table 2.1: An example execution of a Lustre node.

i	1	2	3	4	...
in_i	1	1	1	2	...
out_i	1	$1 + 2 * 1 = 3$	$1 + 2 * 3 = 7$	$2 + 2 * 7 = 16$	...

expressions and have no influence on the program semantics. The body of the node consists of an equation list between keywords `let` and `tel`.

The `pre` operator allows referring to a value at step $n - 1$ in step n. The `->` (followed by) operator allows to initialize streams, which are equal to the value on the left-hand side of the operator at the first instance, and then always equal to the value on the right-hand side.

We can mathematically interpret the computation of output variables as

$$out_i = \begin{cases} in_1 & \text{if } i = 1, \\ in_i + 2 * out_{i-1} & \text{otherwise.} \end{cases}$$

which means the value of *out* is equal to the value of *in* at step 1, then the value of *out* in step s is calculated based on the value of *in* at step i and the value of *out* at the previous step $i - 1$.

An execution result example of the node is shown in Table 2.1. □

Basic value types of Lustre include reals (`real`), integers (`int`), and Booleans (`bool`). In the body of a Lustre node, equations are used to define outputs and values of local variables: $x = e$, where x is a variable and e is a Lustre expression. The equation indicates that the value of x is always equal to e. Expressions in Lustre are made of variable identifiers, constants, arithmetic, Boolean operators, conditional operators, and two operators pre and `->`. For example, the equation $x = y + z$ means that at step i, $x_i = y_i + z_i$. Or the equation $x = c$ where c is a constant means that $x_i = c$ for every step i.

2.4 LTL and assume-guarantee contracts

2.4.1 Linear temporal logic

Temporal logic is used to reason about properties that change over time. The *linear tempraral logic (LTL)* [16] is a type of temporal logic that deals with linear time, where time is considered a sequence of discrete steps.

There are two key operators in temporal logic (we assume φ is an LTL formula):

- The *always operator* $\Box$. $\Box\varphi$ holds iff φ is true in every step. These formulas represent *safety properties*. Safety properties are concerned with the absence of unsafe behaviors to prevent violations or errors within the system.
- The *eventually operator* $\diamond$. $\diamond\varphi$ holds iff φ is true in some steps. These formulas represent *liveness properties*. Liveness properties specify that something good eventually happens during the system's execution.

2.4.2 Assume-guarantee contracts

Compositional reasoning [12] is a technique to improve the scalability of verification tools. It allows breaking a large system into smaller components and analyzing each component separately. The basic idea is that the correctness of a system can be checked by verifying the correctness of its individual components, and then composing those components by analyzing the interactions between them to ensure that the overall system is correct. When performing compositional reasoning on a parent node, the descriptions of child nodes are abstracted by their contracts.

An *assume-guarantee contract* is a pair of a set of *assumptions* A and *guarantees* G. We assume A and G are LTL safety properties. We consider that a target (Lustre) program is annotated with contract (A, G), and represent an annotated program as a *triple*

$$\{A\}\ P\ \{G\}.$$

We say that a triple is *correct* if every execution of P satisfies the LTL formula

$$\Box(\Box A \rightarrow \Box G),$$

that is, the fact that A always holds then G always holds is an invariant.

Examples of assume-guarantee contracts will be described in Section 2.5.1

2.5 Kind 2

Kind 2[2] [4] is an SMT-based MC tool for Lustre programs annotated with CoCoSpec (Section 2.5.1). Kind 2 can verify a given Lustre program P in various ways. First, it can check if P satisfies the invariance of a property φ, which is a Boolean Lustre expression (that can encode safety properties). Kind 2 verifies the invariance for all inputs or outputs a counterexample if it is falsified. Second, Kind 2 can be fed a triple $\{A\}\ P\ \{G\}$, where A and G are sets of Boolean Lustre expressions. Then, it verifies the correctness of the triple.

In addition to *monolithic* mode, Kind 2 provides a *compositional verification* function (Section 2.5.2). When performing verification, Kind 2 may encounter some limitations in handling Lustre programs, such as when the size of the encoded logical formulas increases and the time required for the verification process may become large. In such a case, compositional verification can be performed more efficiently. In the compositional mode, the input is given as a set of triples representing a tree-structured system. The whole process is performed in a bottom-up fashion. First, Kind 2 verifies the leaf triples individually. Then, it verifies the higher-level triples, where child nodes are abstracted with their contracts.

Users can plug in various SMT solvers for Kind 2. Z3 is the recommended SMT solver and the default option. Kind 2 supports several encoding methods (BMC, k-induction, IC3, etc.) and runs several MC processes in parallel and in cooperation.

Example 3. We consider a simple program:

```
node Sample() returns (out: int);
let
  out = 0 -> pre out + 1 ;
  --%PROPERTY out > 0;
tel
```

The annotation `--%PROPERTY` followed by a Boolean Lustre expression is used to specify an invariant property to verify in the node. Kind 2 performs the BMC to falsify the invariance of the property and provides a counterexample that is a signal for *out* whose initial value is 0. If we modify the property to `out >= 0`, Kind 2 is able to prove its invariance by k-induction. □

[2] https://kind2-mc.github.io/kind2/

2.5.1 CoCoSpec

The *CoCoSpec language* [15] supports annotating safety properties in Lustre programs. CoCoSpec extends Lustre to give assume-guarantee contracts to Lustre nodes. Contracts can be declared inline, external, or stand-alone.

We consider the internal contract as special comments which is added inside the declaration of the node. The syntax of an inline contract is:

```
(*@contract
   assume   <Boolean Lustre expression> ;
   guarantee <Boolean Lustre expression> ;
*)
```

Example 4. The code below shows a simple example of an assume-guarantee contract.

```
node N1 (in: bool) returns (out: bool)
(*@contract
  assume in;
  guarantee not out;
*)
let
  out = not in;
tel
```

The node `N1` is fed a `bool` stream *in* and produces another `bool` stream *out*, in which the value of *out* is calculated as the negation of *in* for each step. It is easy to see that if we assume *in* is always true then *out* is always false. Therefore, the annotated node `N1` is correct. □

Example 5. We consider a more complex example.

```
node Integrator (in : real) returns (out : real);
(*@contract
  assume 0.0 <= in and in <= 1.0 ;
  guarantee 0.0 <= out and out <= 9.9 ;
*)
let
  out = in -> (in + 0.9 * pre out);
tel
```

We assume every value of input stream *in* is always greater than or equal to 0 and less than or equal to 1 simultaneously, while the corresponding output *out* is always greater than or equal to 0 and less than or equal to 9.9.

In this case, the guarantee does not hold, it can be proved by BMC that the value of *out* will be greater than 9.9 in an execution step (Example 1). Therefore, the annotated node `Integrator` is incorrect.

If we modify the guarantee to `0.0 <= out and out <= 10.0`, allowing the output value to be 10, Kind 2 can verify its invariance with k-induction. Therefore, the annotated node is verified to be correct. □

2.5.2 Compositional verification in Kind 2

Kind 2 takes the compositional approach by allowing the declarations of assume-guarantee contracts for each node and using reasoning techniques: *compositional reasoning* and *modular reasoning*.

Compositional reasoning in Kind 2 involves the analyzing of the top-level node by abstracting all calls to the sub-nodes. Every node, whose contract contains at least one guarantee, is abstracted by their contracts, then Kind 2 will verify the abstract system. Since the contract has fewer states in comparison to the specified node, compositional reasoning enhances the scalability of Kind 2 by using the information provided by users to reduce the complexity.

Modular reasoning, on the other side, refers to the ability to verify every Lustre node in the hierarchy, bottom-up, without considering the dependencies with other nodes. It allows for the analysis of each node independently, focusing on local properties and contracts. Once the verification is complete for a node, the results can be reused and integrated into the overall verification process of the system.

Only successful results from compositional reasoning are not enough to ensure the correctness of the concrete system since the sub-nodes have not been verified. Therefore the combination of compositional reasoning and modular reasoning is necessary when Kind 2 performs compositional verification. The process can be described as follow:

1. Individual node verification: Kind 2 begins by verifying every Lustre node independently. It analyzes the contract of each node, checking if the assumptions hold under all possible executions and if the guarantees hold. This step ensures that each node satisfies its contract.

2. Dependencies verification: After verifying individual nodes, Kind 2 proceeds to verify the consistency of the system. It considers the interactions between nodes, analyzing how the assumptions and guarantees

of one node relate to the assumptions and guarantees of its dependent nodes.

3. Incremental verification: Kind 2 performs verification incrementally by adding one node context, starting with the top-level node in the system, and verifies it with the abstract calls to its sub-nodes. Once the verification process of the top-level node has been completed, Kind 2 continues with the next level of interconnected nodes until all nodes have been analyzed.

4. Counterexample generation or safety conclusion: If any violations are detected during the verification process, Kind 2 provides a counterexample. The counterexample supports identifying the cause of the violation and helps users to debug the program. If the correctness of all systems in the hierarchy is proved, then the system as a whole is concluded to be safe.

5. Refinement: This step may occur when Kind 2 found a counterexample. The counterexample might be spurious for the concrete system, as a sub-node is abstract, the failure may not happen if we used the concrete one. In this case, if invalid sub-nodes were analyzed and proved correct, Kind 2 will try to refine the call by undoing the abstraction and using the implementation, or the body of those nodes in a new analysis. This failure points out that the system or specification may contain some problems. If the system is proved correct after refining, we should check the contract of invalid sub-nodes and try to perform compositional verification again until the result succeeds without using refinement.

Example 6. We consider an example to compare verification methods.

```
node Integral (Iin: real) returns (Iout: real);
(*@contract
  guarantee Iout <= 9.9 ;
*)
let
  Iout = Iin -> (Iin + 0.9* pre Iout);
tel

node TopLevel (in: real) returns (out: int);
(*@contract
  assume 0.0 <= in and in <= 1.0 ;
```

```
  guarantee out = 1 ;
*)
let
  out = if (Integral(in) <= 9.9) then 1 else 2;
tel
```

In the program, node `TopLevel` calls to `Integral` and produces output values based on the output value of `Integral` for each step. If the output value of `Integral` is less than or equal to 9.9, the output value of `TopLevel` is 1, otherwise, the output value of `TopLevel` is 2. We use Kind 2 to verify whether the output of `TopLevel` is always equal to 1 for every value of the real-number input stream ranging from 0 to 1. We call `Iout <= 9.9` a safety property of node `Integral`, while `out = 1` a safety property of `TopLevel`.

First, we performed monolithic verification. Kind 2 skipped the contract of `Integral` and used the BMC method to prove that the value of *out* will be greater than 9.9 after 43 execution steps, and the safety property `out = 1` was violated.

Then, we performed only compositional reasoning. Kind 2 abstracted the call from `TopLevel` to `Integral` by the contract of `Integral` and skipped the body of the sub-node. In this method, the property of `Integral` is assumed to be invariant, and only the property of `TopLevel` was checked. Because the guarantee of `Integral` ensures that its output is always less than or equal to 9.9, then the Boolean Lustre expression $(integral(in) <= 9.9)$ is always true and leads to the value of *out* is always 1. Hence, the property of `TopLevel` is invariant.

After that, we performed only modular reasoning. Kind 2 started with the verification for node `Integral`. It provided a counterexample in which the input value Iin is 10.9 at the first execution step, then the output value would be 10.9 and violated the guarantee. The safety property of a sub-node was violated. Kind 2 then ran the refinement to check the concrete node and the result was as same as the monolithic verification process. In this method, both safety properties are violated.

Finally, we performed both compositional reasoning and modular reasoning. The result illustrated that the node `Integral` did not satisfy its contract (as the result of the modular reasoning process), while the node `TopLevel` satisfied its contract since the call to the sub-node

was abstracted (as same as compositional reasoning). In general, the safety property of `Integral` is violated, and the property of `TopLevel` is invariant. □

Chapter 3

Formalization

This chapter shows the formalization of the proposed method. We introduce the basic synchronous model, including definitions of synchronous modules, implementation relation, compatibility, and parallel composition operation. Then we describe the interpretation from a Lustre node to a synchronous module.

3.1 Basic synchronous model

Typed variables

Types include `bool`, `int` and `real`, referring to $\{\text{true}, \text{false}\}$, $\mathbb{Z}$ and $\mathbb{R}$, respectively. Typed variables construct *typed expressions*.

Synchronous modules

We consider a set of synchronous modules [8, 10, 11]. A *module* of name m is a tuple $M_m = (I_m, O_m, S_m, \mathit{Init}_m, \mathit{React}_m)$ where each of I_m, O_m, and S_m is a set of input/output/state variables ($I_m \cap O_m \cap S_m = \emptyset$), Init_m is an *initial condition description*, and React_m is a *reaction description*. We denote the domains of I_m, O_m and S_m by $D(I_m)$, $D(O_m)$ and $D(S_m)$, respectively. Init_m is interpreted as a subset of $D(S_m)$.

We have a *reaction* in $D(S_m) \times D(I_m) \times D(O_m) \times D(S_m)$ by an interpretation of React_m; and an *execution* of a module (of length k) is formalized

as a *stream* i.e. a sequence of reactions

$$s_{-1} \xrightarrow{i_0/o_0} s_0 \xrightarrow{i_1/o_1} s_1 \cdots s_{k-2} \xrightarrow{i_{k-1}/o_{k-1}} s_{k-1},$$

where $s_j \in D(S_m)$, $i_j \in D(I_m)$ and $o_j \in D(O_m)$ and s_{-1} satisfies $Init_m$ ($j \in \{-1, \ldots, k-1\}$). A sequence of values $(i_0/o_0 \cdots i_{k-1}/o_{k-1})$ taken from execution is called a *trace*.

Example 7. We model a counter with a "reset" mode as a synchronous module $M_{\texttt{Cnt}}$ with:

- $I_{\texttt{Cnt}} = \{\texttt{reset}\}$,
- $O_{\texttt{Cnt}} = \{\texttt{C}\}$,
- $S_{\texttt{Cnt}} = \{pre(\texttt{C})\}$,
- $Init_{\texttt{Cnt}} \equiv (pre(\texttt{C}) = 0)$, and
- $React_{\texttt{Cnt}}$ is described as

$$\texttt{C} = pre(\texttt{C})' = \begin{cases} 0 & \text{if } \texttt{reset} = \text{true}, \\ 1 + pre(\texttt{C}) & \text{otherwise.} \end{cases}$$

Here, the state variable $pre(\texttt{C})$ represents "the value of C in the previous round." The updated $pre(\texttt{C})'$ in the reaction will be used on the right-hand side in the next round. As an example of executions of $M_{\texttt{Cnt}}$, when an input stream (false false false true) is fed, the output is (0 1 2 0). □

Implementation relation

From the formalization in [10], the implementation relation and the parallel composition mechanism are described as follows. We say a module M_1 *implements* a module M_2 if

1. $O_2 \subseteq O_1$,
2. $I_2 \subseteq I_1 \cup O_1$,
3. A dependency in the evaluation of $x \in I_2 \cup O_2$ on $y \in I_2$ is preserved in M_1, and

4. For every trace t of M_1, the projection of t onto O_2 is a trace of M_2.

We denote this relation by $M_1 \models M_2$. The implementation relation is reflexive ($M_1 \models M_1$), and transitive ($M_1 \models M_2$ and $M_2 \models M_3$ then $M_1 \models M_3$ is obtained).

Compatibility

Given two modules $M_1 = (I_1, O_1, S_1, \mathit{Init}_1, \mathit{React}_1)$ and $M_2 = (I_2, O_2, S_2, \mathit{Init}_2, \mathit{React}_2)$. We say two modules M_1 and M_2 are *compatible* if

1. Outputs of the two modules are disjoint: $O_1 \cap O_2 = \emptyset$,
2. $\mathit{React}_1 \cup \mathit{React}_2$ is acyclic.

Parallel composition

Given two compatible modules $M_1 = (I_1, O_1, S_1, \mathit{Init}_1, \mathit{React}_1)$ and $M_2 = (I_2, O_2, S_2, \mathit{Init}_2, \mathit{React}_2)$. We consider the *parallel composition* of M_1 and M_2, denoted $M_1 \,||\, M_2 = (I, O, S, \mathit{Init}, \mathit{React})$, where $O = O_1 \cup O_2$, $I = (I_1 \cup I_2) \setminus O$, $S = S_1 \cup S_2$, $\mathit{Init} = (\mathit{Init}_1, \mathit{Init}_2)$ and $\mathit{React} = \mathit{React}_1 \cup \mathit{React}_2$. This operation combines two component modules and captures their reactions into a single module. From the definition, commutativity and associativity of the composition operator follow.

- *Commutativity*: $M_1 \,||\, M_2 = M_2 \,||\, M_1$
- *Associativity*: $(M_1 \,||\, M_2) \,||\, M_3 = M_1 \,||\, (M_2 \,||\, M_3)$

Example 8. We consider two modules M_m where $n \in \{1, 2\}$, with $I_m = \{in_m\}$, $O_m = \{out_m\}$, and React_m is described as $out_m = \mathit{not}\ in_m$. Each module receives a `bool` input stream and outputs the negation values. We consider the module M_3 with $I_3 = \{in_1, in_2\}$, $O_3 = \{out_1, out_2\}$, and React_3 is $out_1 = \mathit{not}\ in_1$ and $out_2 = \mathit{not}\ in_2$. Then M_3 represents the *parallel composition* of M_1 and M_2, denoted $M_3 = (M_1 \,||\, M_2)$.

By the definition of implementation relation, we also have $M_3 \models M_1$ and $M_3 \models M_2$. □

3.2 Lustre nodes interpretation

Lustre programs can be naturally interpreted as synchronous modules. Each Lustre node is corresponded with a synchronous module. Then, the input and output variables of the Lustre node are represented by those of the synchronous module. Samewise, the body description is interpreted as the reaction description.

In interpreting reactions, we first expand the right-hand side of the local variables within the expressions in the `let` block. Next, state variables are identified based on the `pre` expressions.

Example 9. We consider a Lustre node:

```
node Sample (in: real) returns (out: bool)
var c, sum: real;
let
  c = 0.0 -> pre sum ;
  sum = in -> in + 0.9*c ;
  out = sum > 9.9 ;
tel
```

The above node is interpreted as a module $(I_{\mathtt{S}}, O_{\mathtt{S}}, S_{\mathtt{S}}, \mathit{Init}_{\mathtt{S}}, \mathit{React}_{\mathtt{S}})$, where:

- $I_{\mathtt{S}} = \{\mathtt{in}\}$,
- $O_{\mathtt{S}} = \{\mathtt{out}\}$,
- $S_{\mathtt{S}} = \{\mathit{pre}(\mathtt{sum})\}$,
- $\mathit{Init}_{\mathtt{S}} \equiv (\mathit{pre}(\mathtt{sum}) = 0)$, and
- $\mathit{React}_{\mathtt{S}}$ represents

$$\mathit{pre}(\mathtt{sum})' = \mathtt{in} + 0.9 \times \mathit{pre}(\mathtt{sum}), \qquad \mathtt{out} = \mathit{pre}(\mathtt{sum})' > 9.9.$$

.

Note that the values of `c` and `sum` are local to an evaluation of the reaction at a round, therefore we do not need to regard them as variables. □

Chapter 4

Proposed method

In this chapter, we propose a new method for checking the correctness of Lustre programs annotated with assume-guarantee contracts (Section 4.1). One of the advantages of our method (summarized in Section 4.2) is that we can handle circular programs. The method translates Lustre nodes and contract properties into synchronous modules (Section 4.3) and then constructs a proof tree based on the deduction rules (Section 4.4).

4.1 Problem statement

The input of our method is an annotated Lustre program P formalized as a set of n triples

$$\{\{A_1\}\ N_1\ \{G_1\}, \ldots, \{A_n\}\ N_n\ \{G_n\}\},$$

where each triple consists of the following items:

- For $1 \leq i \leq n$, N_i is a Lustre node. We assume that P is structured as a tree with a *top-level node* N_n. Nodes $N_1, \ldots, N_{i-1}$ are *sub-nodes* including some *leaf nodes*. A parent node N invokes a child node N' within its body description. Leaf nodes do not have a child.

- A_i is a Boolean Lustre expression on the input variables and previous output values of N_i, which is given in the `assume` section of a CoCoSpec contract.

- G_i is a Boolean Lustre expression on the output variables of N_i, which is given in the CoCoSpec `guarantee` section.

The method will output either of the following two answers:

- "*Correct*" when all the triples are verified to be correct. A triple $\{A_i\}\ N_i\ \{G_i\}$ is *correct* if N_i satisfies the contract, i.e. N_i satisfies the temporal property $\Box(\Box A_i \rightarrow \Box G_i)$ where $\Box$ is the LTL "always" operator (Section 2.4).

- "*Unknown*" when the method fails to verify any of the triples. The method answers unknown when we verify a leaf node using Kind 2 and it fails with a counterexample. Also, it results in an unknown answer without a counterexample when it fails to construct a proof tree.

Example 10. We consider an annotated Lustre program in Figure 4.1 as an input. The input program P consists of three nodes N_1, N_2 and N_3 named as `n1`, `n2` and `toplevel`. A_1 and G_1 are the Boolean Lustre expressions `s1` and `s2` meaning that the input and output are constant true signals. For `toplevel`, we verify that `s2` always holds assuming always true. The output is "correct" because $\{A_1\}\ N_1\ \{G_1\}$, $\{A_2\}\ N_2\ \{G_2\}$, and $\{A_3\}\ N_3\ \{G_3\}$ are all correct. □

4.2 Process of the method

The basic process of the proposed method is as follows:

1. *Translate Lustre nodes into synchronous modules.* First, we analyze the tree structure of nodes $N_1, \ldots, N_n$ and consider a set of node instances $NI_1, \ldots NI_m$, where. $m \geq n$. There can be several instances of a node if it is invoked several times from parent nodes. Then, we interpret each triple instance $\{AI_i\}\ NI_i\ \{GI_i\}$ as an implementation relation $M(NI_i)\ ||\ M(AI_i) \models M(GI_i)$, where $M(NI_i)$, $M(AI_i)$ and $M(GI_i)$ are synchronous modules that correspond to NI_i, AI_i and GI_i. We carefully translate from a triple instance to an implementation relation so that they become equivalent. Note that we also translate properties into modules. Details will be described in Section 4.3.

2. *Construct and validate a proof tree*: We construct a proof tree whose nodes are
$$M(NI_i)\ ||\ M(AI_i) \models M(GI_i),$$

```
node n1 (s1: bool) returns (s2: bool)
(*@contract
  assume s1;
  guarantee s2;
*)
let
  s2 = s1;
tel

node n2 (s2: bool) returns (s1: bool)
(*@contract
  assume s2;
  guarantee s1;
*)
let
  s1 = true -> pre s2;
tel

node toplevel (_: bool) returns (s2: bool)
(*@contract
  guarantee s2;
*)
var s1: bool;
let
  s2 = n1(s1);
  s1 = n2(s2);
tel
```

Figure 4.1: Example of annotated Lustre program containing circular referencing.

where $1 \leq i \leq m$. In the following, we abbreviate them as

$$TN_i \ \equiv \ (M(NI_i) \ || \ M(AI_i) \models M(GI_i)).$$

The top-level node instance TN_m is the root (goal). Here, for simplicity, we assume that TN_i $(1 \leq i \leq m-1)$ are all leaves of the proof tree. The proof is based on the deduction rules for the compositional reasoning of synchronous modules. The tree is constructed as follows:

(a) We verify the correctness of each leaf TN_i $(1 \leq i \leq m-1)$ by using an existing model checking method (Kind 2). If the checking for a leaf fails, the process terminates.

(b) We search for a proof tree using an SMT solver (Z3Py). We encode deduction rules, module declarations, and the fact that TN_i holds for $1 \leq i \leq m-1$ in the predicate logic. Then, by checking the satisfiability of $\neg TN_m$, we can check whether there exists a proof tree.

If not all nodes are leaves, we can repeat the above process for each sub-tree whose root is an intermediate node TN_i where $i < m$. Details will be described in Section 4.4.

Example 11. Example 10 consists of the three Lustre node instances

$$\{AI_1\} \ NI_1 \ \{GI_1\}, \ \{AI_2\} \ NI_2 \ \{GI_2\}, \ \{AI_3\} \ NI_3 \ \{GI_3\},$$

corresponding to `n1`, `n2` and `toplevel`, respectively. The triples can be translated as the following implementation relations:

$$\begin{aligned} TN_1 \ &\equiv \ M(NI_1) \,||\, M(AI_1) \models M(GI_1), \\ TN_2 \ &\equiv \ M(NI_2) \,||\, M(AI_2) \models M(GI_2), \\ TN_3 \ &\equiv \ M(NI_3) \models M(GI_3). \end{aligned}$$

Since $AI_3 \equiv$ true, $M(AI_3)$ is omitted. Also, since $M(AI_1) = M(GI_2)$, we denote them by P_1, and since $M(AI_2) = M(GI_1) = M(GI_3)$, we denote them by P_2. The implementation relations now become:

$$\begin{aligned} TN_1 \ &\equiv \ M(NI_1) \,||\, P_1 \models P_2, \\ TN_2 \ &\equiv \ M(NI_2) \,||\, P_2 \models P_1, \\ TN_3 \ &\equiv \ M(NI_3) \models P_2. \end{aligned}$$

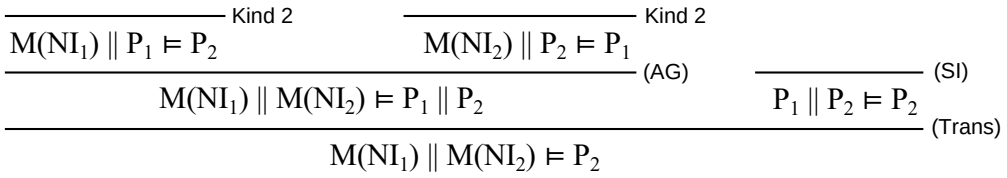

Figure 4.2: An example deductive proof tree.

TN_1 and TN_2 can be verified to be correct using Kind 2. Then, the correctness of TN_3 can be proved as in Figure 4.2. Here, we additionally assume that $M(NI_3) = M(NI_1) \,||\, M(NI_2)$. The existence of the proof tree can be checked using Z3Py. □

By following these steps, our proposed method enables us to verify the correctness of an annotated Lustre program including circular ones. This is different from the compositional verification of Kind 2 as we regard Lustre node instances as synchronous modules and exploit the dedicated deduction rules for circular reasoning. The soundness of the method is followed from the correctness of the translation between Lustre nodes and synchronous modules and the soundness of the deduction rules (Section 4.5).

4.3 Compositional programs interpretation

We interpret both input Lustre nodes and annotated properties into synchronous modules in a way the implementation relation on the modules implies the correctness of the original triple.

4.3.1 Node instances interpretation

In the proposed method, we instantiate nodes invoked by the top-level node (which is also instantiated) and then interpret them as synchronous modules as described in Section 3.2.

Instantiation of nodes can be done by parsing the body description of the nodes and traversing the node invocations from the top-level. For each node invocation, we replace the input and output variables of the called node with the argument expressions (or fresh variables assigned appropriately by the

caller). As a result, the namespace of input and output variables is shared by all node instances (hereafter, we also refer to node instances as nodes).

We represent the translation from node instances to modules as a function M, which applies an interpretation of a Lustre node (instance) as described in Section 3.2.

Example 12. We consider a Lustre program consisting of a parent node `TopLevel` and a sub-node `Filter` (Figure 4.3). `Filter` is instantiated twice by `TopLevel`. We call these two instances `f1` and `f2`. The input and output variables of $M(\texttt{f1})$ and $M(\texttt{f2})$ are as follows:

$$I_{\texttt{f1}} = \{\texttt{pre\_b2}, \texttt{s2}\}, \qquad O_{\texttt{f1}} = \{\texttt{b1}, \texttt{s1}\},$$
$$I_{\texttt{f2}} = \{\texttt{b1}, \texttt{in}\}, \qquad O_{\texttt{f2}} = \{\texttt{b2}, \texttt{s2}\}.$$

□

Although standalone Lustre nodes (i.e. leaf nodes) are easy to interpret, composite node instances NI_m that call other nodes $NI_1, \ldots, NI_{m-1}$ require a special process. Since it is not always the case that $M(NI_m) = M(NI_1) \,||\, \cdots \,||\, M(NI_{m-1})$, we introduce a node instance NR that represents the reaction of NI_m other than calling the child nodes. Accordingly, we interpret NI_m as

$$M(NI_m) \;=\; M(NI_1) \,||\, \cdots \,||\, M(NI_{m-1}) \,||\, M(NR).$$

A *rest node instance* generated from a node instance NI_m consists of the same elements as NI_m, but additionally

- has input variables that are corresponded with the output variables of the child nodes ($TN_1, \ldots, TN_{m-1}$),
- has output variables that are corresponded with the input variables of the child nodes, and
- has reaction equations to assign (i) the outputs of the child nodes to the additional input variables and (ii) the expressions inputted to the child nodes to the additional output variables.

Example 13. In Example 11, we have simply interpreted as $M(NI_3) = M(NI_1) \,||\, M(NI_2)$ without a rest node. A Lustre program in Figure 4.4 illustrates the interpretation with a rest node for this example. The rest

```
node Filter (bin : bool; in : real) returns (bout : bool; out
    : real)
(*@contract
  assume bin;
  assume -1.0 <= in and in <= 1.0;
  guarantee bout;
  guarantee -1.0 <= out and out <= 1.0 ; -- valid (k=25)
*)
var sum, D1, D2: real;
let
  bout = bin;
  sum = 0.0582*(if bin then in else -in) - (-1.49*D1) -
      0.881*D2;
  D1 = 0.0 -> pre sum;
  D2 = 0.0 -> pre D1;
  out = (sum - D2) / 1.25;
tel

node Toplevel (in : real) returns (s1, s2 : real)
(*@contract
  assume -1.0 <= in and in <= 1.0;
  guarantee -1.0 <= s1 and s1 <= 1.0;
*)
var b1, b2, pre_b2 : bool;
let
  b1, s1 = Filter(pre_b2, s2);
  b2, s2 = Filter(b1, in);
  pre_b2 = true -> pre b2;
tel
```

Figure 4.3: An example Lustre node instantiation.

node `rest` for `toplevel` operates as an identify function, which takes the outputs of `n1` and `n2` as input, and outputs the inputs. Figure 4.5 shows the operation diagrams of the original program and modified program.

We also generate a contract for `rest` automatically. Assumptions are inherited from the guarantees of `n1` and `n2`. Likewise, the guarantees are inherited from the assumptions of the child nodes. Since the input and output signals are identical for this example, the assumptions and the guarantees become equivalent, and thus `rest` satisfies the contract. □

4.3.2 Properties interpretation

We interpret Boolean Lustre expressions φ as synchronous modules $M(\varphi)$. We give the function M such that, when applied to AI, NI and GI in a given triple instance $\{AI\}\ NI\ \{GI\}$, the following holds:

$$\{AI\}\ NI\ \{GI\} \text{ is correct} \quad \leftrightarrow \quad M(AI)\,||\,M(NI) \models M(GI)$$

We denote the set of variables in φ by $var(\varphi)$. Then, the output variable set of the module is $O_\varphi = var(\varphi)$. We assume I_φ contains all input variables in the program such that the variables in $var(\varphi)$ depend on. We do not explain how to construct the module $M(\varphi)$ but it should behave as follows:

- Basically, $M(\varphi)$ should output any signal for $var(\varphi)$ that satisfies $\Box\varphi$.
- Also, $M(\varphi)$ should behave the same as the target Lustre program P as long as the behavior of P can satisfy $\Box\varphi$. P is NI when $\varphi \equiv GI$ and P is the external nodes when $\varphi \equiv AI$.
- On the other hand, when P cannot satisfy $\Box\varphi$ due to the valuation of the input variables, $M(\varphi)$ should output an arbitrary signal that still satisfies $\Box\varphi$; therefore, the output of P and $M(\varphi)$ must be different.

For example, we can create $M(\varphi)$ from a composite of NI and a monitor for $\Box\varphi$.

Example 14. In Example 10, node (instance) `n1` assumes the invariance of `s1` being true. We denote this property by p_1. The interpreted module $M(p_1)$ only outputs the constant true signal. □

```
node n1 (s1_: bool) returns (s2: bool)
(*@contract
  assume s1_;
  guarantee s2;
*)
let
  s2 = s1;
tel

node n2 (s2_: bool) returns (s1: bool)
(*@contract
  assume s2_;
  guarantee s1;
*)
let
  s1 = true -> pre s2;
tel

node rest (s1: bool; s2: bool) returns (s1_: bool; s2_: bool)
(*@contract
  assume s1;
  assume s2;
  guarantee s1_;
  guarantee s2_;
*)
let
  s1_ = s1;
  s2_ = s2;
tel

node toplevel (_: bool) returns (s2: bool)
(*@contract
  guarantee s2;
*)
var s1: bool;
let
  (s2, s1) = rest(n1(s1), n2(s2));
tel
```

Figure 4.4: An annotated Lustre program with node rest.

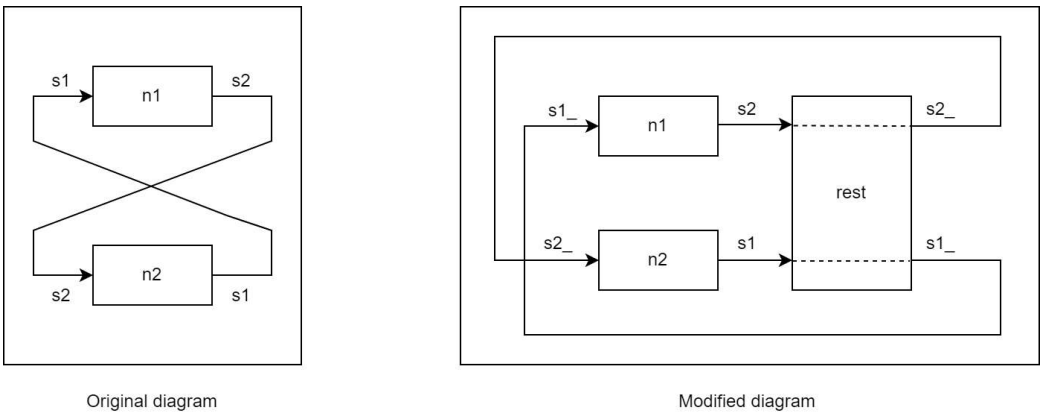

Figure 4.5: Operation diagrams for Example 10.

4.3.3 Triples interpretation

Given a triple $\{A\}\ N\ \{G\}$, we first instantiate N as described in Section 4.3.1. Then, A and G are instantiated by simply substituting the variables as for N. Next, we apply the interpretation function M to each of AI, NI and GI as described in Section 4.3.2. As a result, we obtain a triple instance $\{AI\}\ NI\ \{GI\}$.

Note that $M(NI)$ and $M(AI)$ are compatible so we can always have the parallel composition $M(NI) \,||\, M(AI)$. The relation $M(NI) \,||\, M(AI) \models M(GI)$ will depend on that the output signals of $M(NI) \,||\, M(AI)$ and $M(GI)$ are consistent.

According to the definition of the implementation relation in Section 3.1, the following proposition justifies our method.

Proposition 1. *Consider a triple* $\{A\}\ N\ \{G\}$*, and its instance* $TN \equiv \{AI\}\ NI\ \{GI\}$*. Let* O_{GI} *be the output variable set of* $M(GI)$*. TN is correct iff, for every trace* t *of* $M(AI) \,||\, M(NI)$*, the projection of* t *onto* O_{GI} *is a trace of* $M(GI)$*.*

Proof sketch. Straightforward from the definition of M for the properties (Section 4.3.2). According to whether or not TN is correct, the output trace of $M(AI) \,||\, M(NI)$ onto O_{GI} should be the same as or different from the output of $M(GI)$. □

4.4 Proof tree construction

Given a set of triple instances $TN_1, \ldots, TN_m$, we aim to construct a proof of the correctness of the top-level node instance TN_m (hereafter, we also refer to triple instances as triples). We assume that we know which of the triples becomes a *leaf* in the constructed proof tree (it is determined by the structure of the original program).

To construct a proof tree, we manage a set S of triple instance. It is initially empty. We construct a proof tree with the following steps:

1. We verify the correctness of leaf triples using Kind 2. If successful, append it to S; otherwise, terminate.
2. For each of the unprocessed triples whose child triples are all in S, we construct a proof tree. If successful, append it to S; otherwise, terminate.
3. Repeat Step 2 until TN_m is processed.

Example 15. Assume we have a tree-structured Lustre program P consisting of eight node instances TN_i $(1 \leq i \leq 8)$. Figure 4.6 shows the overall proof tree. It is constructed as follows:

1. We verify TN_1, TN_2, TN_3, TN_4 and TN_7 with Kind 2.
2. We construct a sub-tree for TN_5 with the leaves TN_1 and TN_2.
3. We construct a sub-tree for TN_6 with the leaves TN_3 and TN_4.
4. We construct a sub-tree for TN_8 with the leaves TN_5, TN_6 and TN_7.

If all steps are successful, we output "correct." Otherwise, the process terminates with the output "unknown."

□

4.4.1 Deduction rules for compositional reasoning

In this section, we introduce deduction rules for the construction of a proof tree. The nodes of the proof tree are of the form $M_i \models M_j$ where M_i and M_j $(i, j \in \mathbb{N})$ are synchronous modules. The compatibility of M_i and M_j is denoted $compat(M_i, M_j)$.

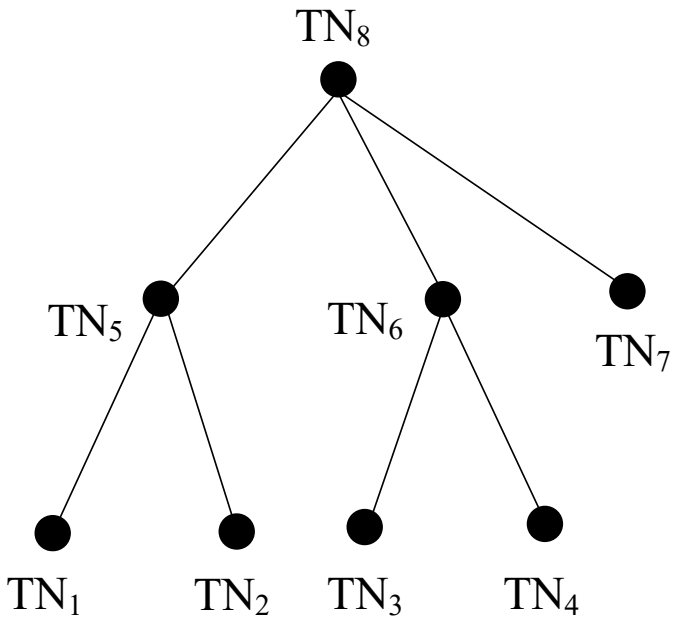

Figure 4.6: An example tree structure of nodes.

- *The assume-guarantee (AG) rule* [10] is a rule for the circular parallel composition of synchronous modules.:

$$\frac{M_1 \,||\, M_3 \;\models\; M_4 \qquad M_2 \,||\, M_4 \;\models\; M_3}{M_1 \;||\; M_2 \;\models\; M_3 \;||\; M_4} \text{ (AG: if } compat(M_1, M_2) \wedge compat(M_3, M_4))$$

  Note that the side condition requires the compatibility of the modules to be composed in the conclusion. The rule has been proved to be sound [10]; namely, the goal node holds whenever the premises and the side condition hold.

- *The sub-node implementation (SI) rules* represent a characteristic of the parallel composition:

$$\frac{}{M_1 \;||\; M_2 \;\models\; M_1} \text{ (SI: if } compat(M_1, M_2))$$

$$\frac{}{M_1 \mid\mid M_2 \models M_2} \text{ (SI: if } compat(M_1, M_2))$$

- *The transitivity rule*:

$$\frac{M_1 \models M_2 \qquad M_2 \models M_3}{M_1 \models M_3} \text{ (Transitivity)}$$

- *The substitution rules*:

$$\frac{M_1 \models M_2}{M_3 \models M_2} \text{ (Substitution: if } M_1 = M_3)$$

$$\frac{M_1 \models M_2}{M_1 \models M_3} \text{ (Substitution: if } M_2 = M_3)$$

  The following equations can be utilized:

  - *Commutativity*: $M_1 \,||\, M_2 = M_2 \,||\, M_1$.
  - *Associativity*: $(M_1 \,||\, M_2) \,||\, M_3 = M_1 \,||\, (M_2 \,||\, M_3)$.

Deduction on the compatibility relation

In parallel to the above deductions, we deduce compatibility relations from initially known ones. The deduction is based on the following rules:

$$compat(M_1, M_2) \wedge compat(M_1, M_3) \wedge compat(M_2, M_3) \rightarrow compat(M_1 \,||\, M_2, M_3),$$
$$compat(M_1, M_2) \wedge compat(M_1, M_3) \wedge compat(M_2, M_3) \rightarrow compat(M_1 \,||\, M_3, M_2),$$
$$compat(M_1, M_2) \wedge compat(M_1, M_3) \wedge compat(M_2, M_3) \rightarrow compat(M_2 \,||\, M_3, M_1),$$
$$compat(M_1, M_2) \leftrightarrow compat(M_2, M_1).$$

4.5 Soundness of the method

The soundness of the proposed method can be stated as follows:

Proposition 2. *Given an annotated Lustre program consisting of triples* $\{A_1\}\ N_1\ \{G_1\}, \ldots, \{A_n\}\ N_n\ \{G_n\}$*, where* N_n *is a top-level node, if the proposed method outputs "correct," then* $\{A_n\}\ N_n\ \{G_n\}$ *is correct.*

Proof sketch. Checking the correctness of $\{A_n\}\ N_n\ \{G_n\}$ means to check that of $\{AI_m\}\ NI_m\ \{GI_m\}$. As described in Section 4.3, the correctness of $\{AI_m\}\ NI_m\ \{GI_m\}$ is equivalent to $M(AI_m)\,||\,M(NI_m) \models M(GI_m)$. When constructing a proof tree, leaf nodes are verified with Kind 2, which should provide a sound process. Based on the verified leaf nodes, deductions are made to prove $M(AI_m)\,||\,M(NI_m) \models M(GI_m)$ based on the rules in Section 4.4.1, each of which is sound. □

Chapter 5

Experiments

This chapter describes the implementation of our proposed method using Kind 2 and Z3Py. We also present examples and experiments to evaluate the method.

Our experiments were conducted on the Surface Pro 2020 with an Intel Core i5-8350U processor and 8 GB of RAM.

5.1 Implementation

We implemented a prototype tool in Python based on the proposed method. The implementation is intended to show how the proposed method works with different examples. By observing and analyzing the results, we are able to evaluate the efficiency of our method.

The tool uses Kind 2 and Z3Py to perform compositional verification on an input program. The input of the tool is not an annotated Lustre program but a Python script that describes a set of triple instances. The implementation consists of:

1. *A translator module.* Given node instances and annotated properties, the module generates an intermediate representation of synchronous modules.

2. *A printer module.* Prints intermediate data, such as a monolithic form of the input program and separated Lustre node instances.

3. *A validator module.* Feeds the definition of the deduction rules, the

declarations of modules, and the compatibility relations among modules to Z3Py and searches for a proof tree using the Z3 solver.

The input Python script consists of:

- A dictionary object `properties` whose keys are the property identifiers and whose values are dictionary objects. A value of the inner dictionary consists of a list `vars` of the variables of the property and a string `expression` of the Boolean Lustre expression (used when printing Lustre programs). These properties are taken from the contracts of the nodes.

- A dictionary object `nodes` whose keys are the identifiers of node instances and whose values are dictionary objects. The inner dictionary consists of lists `iv` and `ov` of input and output variables, another dictionary `contract` containing assume and guarantee properties, and a string `body` of a Lustre code fragment for a `let` block.

- A dictionary object `dependencies` whose keys are the names of node instances and whose values are lists of the names of the (direct) child node instances.

As an output, the tool prints "`correct`" or "`unknown`" according to the result of the verification process.

Example 16. We consider the program in Example 10. Its interpretation as a set of synchronous modules is described in Section 4.2. The `properties` described as a Python script is as follows:

- Dictionary `properties` describes the properties `P1` and `P2`:

```
properties = {
  'p1': {
    'vars': ['s1'],
    'expression': 's1'
  },
  'p2': {
    'vars': ['s2'],
    'expression': 's2'
  }
}
```

- Dictionary `nodes` describes Lustre node instances `n1`, `n2` and `toplevel`, corresponding to $M(NI_1)$, $M(NI_3)$, $M(NI_3)$, respectively:

```
nodes = {
  'n1': {
    'iv': [ { 'name': 's1', 'type': 'bool' } ],
    'ov': [ { 'name': 's2', 'type': 'bool' } ],
    'contract': {
      'assume': ['p1'],
      'guarantee': ['p2']
    },
    'body': '''let
  s2 = s1;
tel'''
  },
  'n2': {
    'iv': [ { 'name': 's2', 'type': 'bool' } ],
    'ov': [ { 'name': 's1', 'type': 'bool' } ],
    'contract': {
      'assume': ['p2'],
      'guarantee': ['p1']
    },
    'body': '''let
  s1 = true -> pre s2;
tel'''
  },
  'toplevel': {
    'is_main': True,
    'iv': [],
    'ov': [ { 'name': 's2', 'type': 'bool' } ],
    'contract': {
        'assume': [],
        'guarantee': ['p2']
    },
    'body': '''var s1: bool;
let
  s2 = n1(s1);
  s1  = n2(s2);
tel'''
  }
}
```

- Dictionary `dependencies` describes that `n1` and `n2` are leaves, and that they are the children of `toplevel`:

```
dependencies = {
  'toplevel': ['n1', 'n2'],
  'n1': [],
  'n2': []
}
```

- The script also contains commands to instantiate the `Verifier` class and to call its method (first, the translator module will be invoked):

```
verifier = Verifier()
verifier.verify(nodes, properties, dependencies)
```

□

5.2 First experiment

We conducted the first experiment to demonstrate the increase of the execution time of MC and to confirm the effectiveness of the compositional approach. In the experiment, we performed a monolithic verification process using Kind 2 and a compositional verification process using our tool.

The example used an integrator delayed by two nested counters. We analyze each part of the program. First, there are two constants: *max* is the limit for the counters, and *threshold* is the limit for the integrator. These two parameters are important as they affect the number of calculations and execution time.

```
const max = 3;
const threshold = 9.9;
```

The output of the first counter increases by 1 after each step. The output of the second counter increases by 1 when the input value `c1` is equal to *max*, otherwise, the output value remains unchanged. Both counters are reset to 0 after reaching the *max* value.

```
node counter1 () returns (c1: int)
(*@contract
  guarantee c1 <= max;
*)
```

```
let
  c1 = 0 -> if pre c1 = max then 0 else pre c1 + 1;
tel

node counter2 (c1: int) returns (c2: int)
(*@contract
  assume c1 <= max;
  guarantee c2 <= max;
*)
let
  c2 = 0 -> if pre c2 = max then 0 else if c1 = max then pre c2 +
       1 else pre c2;
tel
```

The integrator was introduced in Example 5. In this example, its output is only updated when the input value c2 is equal to max.

```
node integral (in: real; c2: int) returns (out: real)
(*@contract
  assume c2 <= max;
  assume -1.0 <= in and in <= 1.0;
  guarantee out < threshold;
*)
let
  out = in -> if c2 = max then (in + 0.9* pre out) else pre out;
tel
```

Finally, the top-level node calls to all the sub-nodes above with a guarantee about the output of the integrator.

```
node toplevel (in: real) returns (out: real)
(*@contract
  assume -1.0 <= in and in <= 1.0;
  guarantee out < threshold;
*)
var c1 : int;
var c2 : int;
let
  c1 = counter1() ;
  c2 = counter2(c1) ;
  out = integral(in, c2) ;
tel
```

Table 5.1: Execution time and analyzed steps required for monolithic verification.

Max \ Threshold	4.0	6.0	8.0	9.0	9.9
1	0.2s k=8	0.3s k=16	0.5s k=30	0.7s k=42	1.6s k=86
2	0.3s k=24	0.6s k=48	1.6s k=90	3.6s k=126	16.4s k=258
4	1.1s k=80	5.4s k=160	22.7s k=300	59.1s k=420	TO
6	2.9s k=168	18.8s k=336	154s k=630	TO	-
8	8.6s k=288	75.6s k=576	TO	-	-
10	34.6s k=440	267s k=880	-	-	-
12	72.1s k=624	TO	-	-	-
14	201s k=840	-	-	-	-
15	260s k=960	-	-	-	-
16	TO	-	-	-	-

The program can be proved to be incorrect if *threshold* is less than 10 (explained in Example 6). We gradually increased those values and observed correlations between them and the execution time as well as the number of analyzed steps. The timeout was set at 300 seconds. The experimental results are showed in Table 5.1 and Table 5.3.

The monolithic process requires the verification tool to execute every step until the property is proved or time out. Since the increase of counter 2 is delayed by counter 1 (for an amount of max steps), and the increase of the integrator is delayed by counter 2 (for an amount of $max \times (max + 1)$ steps), the number of analyzed steps for the integrator to go over the threshold increases along with the values of max and *threshold*. We can see that

Table 5.2: An example execution when performing monolithic verification.

Step	1	2	3	4	5	6	7	8	9	10	11	12
c1	0	1	2	0	1	2	0	1	2	0	1	2
c2	0	0	1	1	1	2	0	0	1	1	1	2
in	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
out	1.0	1.0	1.0	1.0	1.0	1.9	1.9	1.9	1.9	1.9	1.9	2.71

Table 5.3: Execution time and analyzed steps required for compositional verification.

Max \ Threshold	4.0	6.0	8.0	9.0	9.9
any	1.7s k=4	1.9s k=8	6.7s k=15	16s k=21	170s k=43

execution time increased rapidly, possibly up to 16 times for one increase ($threshold = 9.0$ and max increased from 2 to 4). The number of analyzed steps increased significantly, too. As the $threshold$ value approached 10 (but was not equal to 10), the timeout occurred quickly.

For instances, if $max = 2$ and $threshold = 2.0$, the execution is shown in Table. 5.2.

Normally, if the integrator is executed separately, it takes only 3 steps to go over the threshold, while the example above took up to 12 steps. In general, as we increase the values of max and $threshold$, the number of analyzed steps also increases and execution time follows.

The compositional process, on the other hand, took much less time and analyzed step. For instance, 1.7 seconds and 4 steps in comparison to 259 seconds and 959 steps ($threshold = 4.0$ and $max = 15$), which means up to a 150-fold improvement in execution time. This method verified each node individually so the increase of the integrator was not affected by the counters.

5.3 Second experiment

The second experiment was conducted to evaluate whether our implementation verifies the Lustre programs correctly and efficiently. In this experiment,

we verified several circular examples that cannot be handled by the compositional verification function of Kind 2.

Simplified circular program

We considered the Lustre program in Example 10.

As we mentioned before, this example cannot be handled by the compositional verification function of Kind 2. Note that Kind 2 could verify it with the monolithic mode. Although the correctness of sub-nodes N1 and N2 are proved by modular analyzing when verifying the entire system with dependencies, compositional reasoning takes into account the interactions between nodes and generates a spurious counterexample, in which input signals of both sub-nodes are `false` and leads to the output signals are also `false`.

First, we applied our method to the simplified program, in which we assumed that the interpretation module of *toplevel* node is the parallel composition of the leaves (Section 4.3): $M(NI_3) = M(NI_1) \,||\, M(NI_2)$.

The proof tree of the program was given in Section 4.2. Our tool gave the result *correct* after 0.006 seconds (Fig. 5.1).

Circular program

We applied our method to the circular program with node `rest` generated. Detail about node `rest` is introduced in Section 4.3.

Now the program consists of four synchronous modules $M(NI_1)$, $M(NI_2)$, $M(NR)$ and $M(NI_3)$ corresponding to the instance of nodes `n1`, `n2`, `rest` and `toplevel`, respectively. The triples can be translated as the following implementation relations:

$$\begin{aligned}
&M(NI_1) \,||\, M(AI_1) \models M(GI_1),\\
&M(NI_2) \,||\, M(AI_2) \models M(GI_2),\\
&M(NR) \,||\, M(AR) \models M(GR),\\
&M(NI_3) \models M(GI_3).
\end{aligned}$$

We interpreted the Boolean Lustre expression $s1$, $s2$, $s1\_$, $s2\_$ as synchronous modules P_1, P_2, $P_1{}'$, $P_2{}'$, respectively. Then we had

$$M(AI_1) = P1', M(GI_1) = P2, M(AI_2) = P2', M(GI_2) = P1,$$

$$M(AR) = P1 \,||\, P2, M(GR) = P1' \,||\, P2', M(GI_3) = P2.$$

```
Verifying node n1 by Kind 2...
Result: Success.
Verifying node n2 by Kind 2...
Result: Success.
Leaf nodes are verified.
n1 = cv.new_node()
n2 = cv.new_node()
cv.add(compat(n1,n2))
p1 = cv.new_node()
p2 = cv.new_node()
cv.add(compat(p1,p2))
cv.add(compat(p1,n1))
cv.add(impl(pc(n1, p1),p2))
cv.add(compat(p2,n2))
cv.add(impl(pc(n2, p2),p1))
cv.check(impl(pc(n1, n2),p2))
Program verification result: Correct.

time: 0.006000
memory: 17.240000
```

Figure 5.1: Verification result for the simplified circular program.

The implementation relations become:

$$M(NI_1) \,||\, P_1{}' \models P_2,$$
$$M(NI_2) \,||\, P_2{}' \models P_1,$$
$$M(NR) \,||\, P_1 \,||\, P_2 \models P_1{}' \,||\, P_2{}',$$
$$M(NI_3) \models P_2.$$

We constructed the proof tree for the program (Fig 5.2).

Then we verified the program with our tool and got the result *correct* after 0.011 seconds of execution time.

```
...
Program verification result: Correct.

time: 0.011000
memory: 18.200000
```

P_1 s1
P_2 s2
P_1' s1_
P_2' s2_
(1) $(M(NI_1) \parallel P_1') \parallel P_1 \vDash M(NI_1) \parallel P_1'$
(2) $M(NI_1) \parallel P_1' \vDash P_2$
(3) $(M(NI_2) \parallel P_2') \parallel P_2 \vDash M(NI_2) \parallel P_2'$
(4) $M(NI_2) \parallel P_2' \vDash P_1$
(5) $M(NI_1) \parallel P_1' \parallel P_1 \vDash P_2$
(6) $M(NI_2) \parallel P_2' \parallel P_2 \vDash P_1$
(7) $(M(NI_1) \parallel P_1') \parallel (M(NI_2) \parallel P_2') \vDash P_1 \parallel P_2$
(8) $M(NR) \parallel (P_1 \parallel P_2) \vDash P_1' \parallel P_2'$
(9) $(M(NI_1) \parallel M(NI_2)) \parallel M(NR) \vDash (P_1 \parallel P_2) \parallel (P_1' \parallel P_2')$
(10) $(P_1 \parallel P_2) \parallel (P_1' \parallel P_2') \vDash P_1 \parallel P_2$
(11) $(M(NI_1) \parallel M(NI_2)) \parallel M(NR) \vDash P_1 \parallel P_2$
(12) $P_1 \parallel P_2 \vDash P_2$
(13) $(M(NI_1) \parallel M(NI_2)) \parallel M(NR) \vDash P_2$

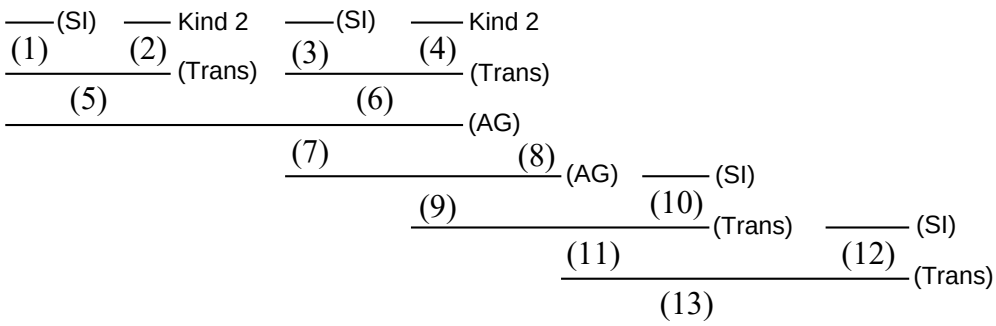

Figure 5.2: Proof tree for the circular program.

Circualar program with intermediate node

In the example above, we handled a program that has one top-level triple instance and the others are all leaves. Now we consider an extended program containing an intermediate node.

The node `toplevel` was renamed to `intermediate`. Then we defined a

node n3 and a new toplevel node which calls to intermediate and n3.

```
node n3 (s2: bool) returns (s3: bool)
(*@contract
  assume s2;
  guarantee s3;
*)
let
  s3 = s2;
tel

node intermediate () returns (s2: bool)
(*@contract
  guarantee s2;
*)
var s1: bool;
let
  s2 = n1(s1);
  s1  = n2(s2);
tel

node toplevel () returns (s3: bool)
(*@contract
  guarantee s3;
*)
var s2: bool;
let
  s2 = intermediate();
  s3  = n3(s2);
tel
```

The dependencies of the input Python script was updated to

```
dependencies = {
  'toplevel': ['intermediate', 'n3'],
  'intermediate': ['n1', 'n2'],
  'n1': [],
  'n2': [],
  'n3': []
}
```

which means the tree structure contains two sub-trees. One sub-tree has

`intermediate` as the root, while `n1` and `n2` are leaves. The other sub-tree has `toplevel` as the root, while `intermediate` and `n3` are leaves.

The verification process follows the description in Section 4.4:

1. The tool verifies all leaf nodes `n1`, `n2` and `n3` with Kind 2. All leaf nodes are verified to be correct and appended to S - a list containing verified nodes.

2. The tool checks that node `intermediate` is not verified and all children of `intermediate` are in S. It continues to verify node `intermediate`. After the verification succeeded, `intermediate` is added to S.

3. The tool checks that node `toplevel` is not verified and all children of `toplevel` are in S. It continues to verify node `toplevel`. After the verification succeeded, `toplevel` is added to S. The verification process is complete.

We performed the verification and got the results *correct* after 0.024 seconds of execution time in total.

```
Verifying node n1 by Kind 2...
Result: Success.
Verifying node n2 by Kind 2...
Result: Success.
All leaf nodes have been verified.

Start verifying new sub-tree:
...
cv.check(impl(pc(rest, pc(n1, n2)),p2))
Verification result for the node intermediate: Correct.
time: 0.008000
memory: 18.200000

Start verifying new sub-tree:
...
cv.check(impl(pc(rest, pc(intermediate, n3)),p3))
Verification result for the node toplevel: Correct.
time: 0.016000
memory: 18.430000
Verification completed.
```

Two digital filters

We conducted an experiment on a complex example containing two digital filters, which was introduced in Example 12.

The program declares a node `Filter` as the sub-node and node `toplevel` as the top-level node. `toplevel` invokes `Filter` twice so there are two instances of filter or two filters with the same functionality.

By applying our tool, we printed out two instances of node `Filter`:

```
node f1 (pre_b2: bool; s2: real) returns (b1: bool; s1: real)
(*@contract
  assume pre_b2;
  assume -1.0 <= s2 and s2 <= 1.0;
  guarantee b1;
  guarantee -1.0 <= s1 and s1 <= 1.0;
*)
var sum, D1, D2: real;
let
  b1 = pre_b2;
  sum = 0.0582*(if pre_b2 then s2 else -s2) - (-1.49*D1) - 0.881*
      D2;
  D1 = 0.0 -> pre sum;
  D2 = 0.0 -> pre D1;
  s1 = (sum - D2) / 1.25;
tel

node f2 (b1: bool; in: real) returns (b2: bool; s2: real)
(*@contract
  assume b1;
  assume -1.0 <= in and in <= 1.0;
  guarantee b2;
  guarantee -1.0 <= s2 and s2 <= 1.0;
*)
var sum, D1, D2: real;
let
  b2 = b1;
  sum = 0.0582*(if b1 then in else -in) - (-1.49*D1) - 0.881*D2;
  D1 = 0.0 -> pre sum;
  D2 = 0.0 -> pre D1;
  s2 = (sum - D2) / 1.25;
tel
```

These two instances also contain circular referencing as each of them provides input for the other. When verifying each instance individually, Kind 2 performed the k-induction method. The process took a lot of time (270 seconds) and 25 inductive steps because the computations inside the node are very complicated. Hence we verified the node `Filter` first and confirmed it was correct. Then we assumed the node has been correct when applying our tool in this program so that we could save time from the verification process of the leaf node.

We constructed a proof tree (Fig. 5.3). The program consists of four synchronous modules $M(NI_1)$, $M(NI_2)$, $M(NR)$ and $M(NI_3)$ corresponding to the instance of nodes `f1`, `f2`, `rest` and `toplevel`, respectively. Denotations of Boolean Lustre expressions are also shown in Fig. 5.3.

The proof tree was not difficult to construct by hand. But when we applied our tool to the program, the result was *unknown* because of the timeout, even though we raised the timeout limitation to 1000 seconds.

We checked the number of premises to construct the proof tree and saw that there were 73 premises generated. This large number of premises created many possible deductions and took a lot of time to search for the correct answers. We tried to find solutions to deal with this problem. We realized that during the construction of the proof tree, there were several properties which are always appeared together in each deduction step, e.g., `pre_b2` and `-1.0 <= s2 and s2 <= 1.0`. Hence, we modified the *translator* module by adding a pre-process to group those properties into a single property. This aimed to reduce the number of modules and the size of the search space.

After the modification, the number of premises decreased to 43. And the result *correct* was provided after 0.311 seconds (without verification of the leaf nodes by Kind 2).

5.4 Discussions

The result of the first experiment shows that monolithic MC may become time-consuming when it requires analyzing a number of execution steps. Compositional verification can be more effective in such cases. As a future work, we can conduct a temporal composition, which splits the execution timeline of a system into several segments and analyzes each of them, instead of separating the system by each component.

In the second experiment, our implementation of the proposed method

P1 pre_b2 and -1.0 <= s2_ and s2_ <= 1.0

P2 b1 and -1.0 <= s1 and s1 <= 1.0

P3 b1_ and -1.0 <= in_ and in_ <= 1.0

P4 b2 and -1.0 <= s2 and s2 <= 1.0

P5 -1.0 <= in_ and in_ <= 1.0

P6 -1.0 <= s1 and s1 <= 1.0

(1) (M(NI1) || P1) || P4 ⊨ (M(NI1) || P1)

(2) M(NI1) || P1 ⊨ P2

(3) (M(NI2) || P3) || P2 ⊨ (M(NI2) || P3)

(4) M(NI2) || P3 ⊨ P4

(5) (M(NI1) || P1) || P4 ⊨ P2

(6) (M(NI2) || P3) || P2 ⊨ P4

(7) (M(NI1) || P1) || (M(NI2) || P3) ⊨ P2 || P4

(8) (M(NI2) || P5) || (P2 || P4) ⊨ P1 || P3

(9) (M(NI1) || M(NI2)) || (M(NR) || P5) ⊨ (P1 || P3) || (P2 || P4)

(10) P1 || P3 || P2 || P4 ⊨ P2

(11) M(NI1) || M(NI2) || M(NR) || P5 ⊨ P2

(12) P2 ⊨ P6

(13) M(NI1) || M(NI2) || M(NR) || P5 ⊨ P6

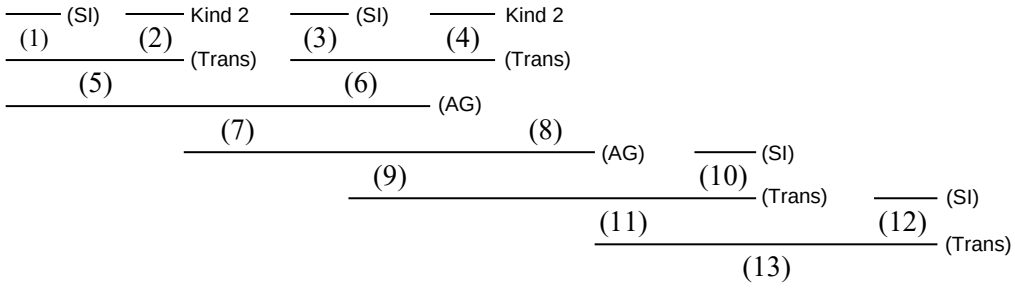

Figure 5.3: Proof tree for the program of two digital filters.

was able to verify circular Lustre programs. We confirmed that the results were correct. With the monolithic mode, Kind 2 is also able to verify simple examples efficiently. With the compositional mode, Kind 2 cannot handle

the examples, resulting in spurious counterexamples.

We identified a scalability issue with our tool. For some examples involving complex contracts, the proof construction process by the validator module becomes time-consuming. We consider that this is due to the increase of the number of equivalent composite module terms and the number of the *compat* terms (then, the number of combinations examined by Z3 will increase). A workaround to handle such cases is to assert implementation relations that are expected to hold in the middle of the proof tree. We confirmed that this workaround actually reduces the execution time of Z3. Improvement of the proof tree construction process is a future work.

Chapter 6

Related work

Alur and Henzinger have proposed the concepts of synchronous modules and the compositional reasoning on them [10]. They also implemented the Mocha tool [19] for the compositional verification of real-time systems. While Mocha requires manual efforts by the users, we aim at an automated method in this research.

Champion et al. have proposed Kind 2 [4] and CoCoSpec [15], which are tools for the compositional verification of Lustre programs as described in Section 2.5. As mentioned earlier, the compositional verification function of Kind 2 has a limitation in handling circular programs.

While most of the research on circular compositional reasoning addresses safety properties, McMillan [7] studied how to reason for liveness properties.

Studies have been conducted to automatically generate contracts by using the L* algorithm for example [12]. In this research, we assume that input programs are annotated appropriately by the users. We can use a contract generation method to facilitate the compositional verification process.

Chapter 7

Conclusion

7.1 Conclusion

Verifying the safety of practical synchronous systems is important and compositional verification is promising since practical programs may be very large and complex. In this thesis, we studied how to apply deduction rules for circular reasoning so that we can perform compositional verification on Lustre programs with a different approach from the Kind 2 tool.

We proposed a new compositional MC function for Lustre programs annotated with an assume-guarantee contract. The method has one advantage is that it can handle circular programs, which is not completely supported in Kind 2. Our approach is also different from the compositional verification of Kind 2 as we translate Lustre nodes and contract properties into the concepts of synchronous modules and exploit the dedicated deduction rules for circular reasoning, then by checking the existence of a proof tree, we are able the verify the correctness of an annotated Lustre program including circular ones.

We implemented a tool as a Python script using Z3Py and Kind 2 to evaluate the efficiency of the proposed method and our implementation. The results showed that the examples of Lustre programs are verified correctly. During the experiment, we discovered that the process may become inefficient due to the large search space among many possible deductions in compositional reasoning.

7.2 Future work

Our proposed method works for *assume-guarantee* contract of CoCoSpec language but has not yet supported *mode* [15]. Therefore, we will try to handle annotated Lustre programs with *mode* contracts.

The experiments that we conducted were still based on simple examples. We need to work on improving the efficiency of the proposed method and applying it to practical Lustre programs.

Besides, we can study a new temporal composition method to verify the Lustre programs with lengthy behaviors.

Finally, the compositional MC function of Kind 2 requires manually annotating the contract. It will be more practical if the annotation process can be automated. In the future, we may study how to generate contracts for infinite domains in Lustre. We consider using learning techniques such as the L* algorithm.

Bibliography

[1] Barrett, C., Tinelli, C.: Satisfiability Modulo Theories. In: Handbook of Model Checking, chap. 11, pp. 305–343 (2018)

[2] Clarke, E. M., Henzinger, T. A., Veith, H.: Introduction to Model Checking. In: Handbook of Model Checking, chap. 1, pp. 1–26 (2018)

[3] Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.A.: LUSTRE: A declarative language for programming synchronous systems. In: POPL. pp. 178–188 (1987)

[4] Champion, A., Mebsout, A., Sticksel, C., Tinelli, C.: The KIND 2 Model Checker. In: Computer Aided Verification, pp. 510—517 (2016)

[5] Nejati, F., Ghani, A. A. A., Yap, N. K., Jafaar, A. B.: Handling State Space Explosion in Component-Based Software Verification: A Review. In: IEEE Access, vol. 9, pp. 77526–77544 (2021)

[6] Nancy, G. L., and Clark S. T.: An Investigation of the Therac-25 Accidents. In: IEEE Computer, pp. 18–41 (1993)

[7] McMillan, K.L.: Circular Compositional Reasoning about Liveness. In: Correct Hardware Design and Verification Methods, pp. 342-–346 (1999)

[8] Halbwachs, N., Lagnier, F., Raymond, P.: Synchronous Observers and the Verification of Reactive Systems. In: Algebraic Methodology and Software Technology (AMAST). pp. 83–96 (1993)

[9] Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous dataflow programming language LUSTRE. In: Proceedings of the IEEE, vol. 79, no. 9, pp. 1305–1320 (2000)

[10] Alur, R., Henzinger, T.A.: Reactive modules. Formal Methods in System Design 15(1), 7–48 (1999)

[11] Alur, R.: Synchronous Model. In: Principles of Cyber-Physical Systems, chap. 2, pp. 13–64. MIT Press (2015), http://mitpress.mit.edu/books/principles-cyber-physical-systems

[12] Giannakopoulou, D., Kedar, S., Corina, S.: Compositional Reasoning. In: Handbook of Model Checking, chap. 12, pp. 345–383 (2018)

[13] Kroening, D., Strichman, O.: Decision Procedures, 2nd ed. Springer, Heidelberg (2016)

[14] Biere, A., Cimatti, A., Clarke, E. M., Strichman, O., Zhu, Y.: Bounded model checking. In: Handbook of satisfiability, pp. 457–481 (2009)

[15] Champion, A., Gurfinkel, A., Kahsai, T., Tinelli, C.: CoCoSpec: A mode-aware contract language for reactive systems. In: SEFM, pp. 347–366 (2016)

[16] Piterman, N., Pnueli, A.: Temporal logic and fair discrete systems. In: Handbook of Model Checking, chap. 2, pp. 27–73 (2018)

[17] de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340 (2008)

[18] Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems. (1995)

[19] Alur, R., Henzinger, T.A., Mang, F.Y.C., Qadeer, S., Rajamani, S.K., Tasiran, S.: MOCHA: Modularity in model checking. In: Computer Aided Verification, pp 521—525 (1998)