INSTITUT FÜR INFORMATIK
Lehrstuhl für Softwaretechnik und Programmiersprachen
Universitätsstr. 1 D–40225 Düsseldorf

Simulation and Verification of Reactive Systems in Lustre with ProB

Fabian Vu

Masterarbeit

Beginn der Arbeit:	04. November 2019
Abgabe der Arbeit:	22. Juni 2020
Gutachter:	Prof. Dr. Leuschel
	Prof. Dr. Klau

Erklärung

Hiermit versichere ich, dass ich diese Masterarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Düsseldorf, den 22. Juni 2020

____

Fabian Vu

Abstract

ProB is an animator, constraint solver, and model checker for the B method. In addition to B, it supports other specification languages such as Event-B, XTL, CSP-M, TLA+, Z, and Alloy. These languages are used in the context of safety-critical systems.

This work extends ProB by the programming language Lustre which is also used for safety-critical systems, especially reactive systems. As a result, this thesis presents a ProB interpreter for Lustre, and a translator from Lustre to B named Lustre2B. Both approaches provide the opportunity to simulate Lustre programs, and verify them using different model checking techniques. When it comes to safety-critical systems, it is not only important to verify the system, but also to reason about its behavior. As achieved in this work, it is possible to view the program's state and to trace the program's execution of Lustre programs in the ProB animator. Lustre2B also targets a subset of B that makes code generation to Java and C++ for simulation using B2Program feasible.

Furthermore, a Lustre parser is implemented in this work that is used by both approaches. The parser is designed to generate, check, and optimize the Lustre AST such that interpretation and translation to B are simplified. It also determines the state of a Lustre program in the ProB animator. Together with the resulting AST, this information is passed to the ProB interpreter and Lustre2B as well.

Finally, this work investigates the performance of simulation according to the ProB interpreter for Lustre, the translated B machines using Lustre2B, and the Java and C++ code generated from B2Program. Furthermore, it is also analyzed how explicit-state model checking and bounded model checking perform on Lustre programs.

Contents

1 Introduction and Motivation

This thesis explores the simulation and verification of LUSTRE programs. The main focus is set on the animator, constraint solver, and model checker PROB.

LUSTRE is mainly used in embedded systems that react to their environment in real-time. When it comes to safety-critical systems, it is not only important to verify the system, but also to reason about its behavior. During this work, it is investigated how LUSTRE can be integrated into PROB. According to verification, this enables different verification techniques such as explicit-state model checking, symbolic model checking, and LTL model checking. Furthermore, it is then possible to view the program's state and to trace the program's execution in the PROB animator. On the one hand, it is possible to access previous values of expression flows in LUSTRE. On the other hand, it is also possible to sample them on other clocks. So, this work also explores how a state in the PROB animator can be represented.

There are two different approaches to extend PROB by LUSTRE: implementing a PROB interpreter for LUSTRE and implementing a translator from LUSTRE to B.
While the first approach makes it possible to interpret a LUSTRE program's AST, the latter approach provides the possibility to load the B models that are translated from LUSTRE in PROB. This is the way how various features of PROB including simulation and verification can be applied to LUSTRE programs. The advantages and disadvantages of both approaches are also outlined in this work. During this work, a LUSTRE parser, a LUSTRE AST interpreter for PROB, and a translator from LUSTRE to B named LUSTRE2B are implemented using the programming language SICStus Prolog. [1] [2] Furthermore, this work also aims translation to a subset of B from which Java and C++ code can be generated using the code generator B2PROGRAM for simulation.

Finally, the performance of both simulation and verification are investigated. There are four possibilities to simulate LUSTRE programs in this work: using the LUSTRE AST interpreter, translation to B, and code generation to Java and C++. The performance of all of them is analyzed and compared in this work. In contrast, the performance of verification is analyzed for explicit-state model checking and bounded model checking (BMC). While explicit-state model checking is supported for both approaches in PROB, BMC is supported for the translated B models only. Furthermore, it is analyzed how good they can detect errors.

2 Background

2.1 B-Method

The B-Method was introduced by Jean-Raymond Abrial and is used to specify and verify software systems, especially safety-critical systems [1]. For example, *Paris Métro Line 14* [2] and *ETCS Hybrid Level 3 Concept* [3] are verified using the B-Method.

[1] `https://gitlab.cs.uni-duesseldorf.de/vu/lustre-interpreter`

[2] Lustre Interpreter integrated in PROB: `https://gitlab.cs.uni-duesseldorf.de/stups/prob/prob_prolog/-/tree/lustre`

The B-Method contains the B specification language which is based on *set theory* and *first-order logic*. A component in B is called a *machine* which contains clauses for sets, variables, constants, invariant, initialization, and operations.
The clauses `SETS`, `VARIABLES`, and `CONSTANTS` declare sets, variables, and constants respectively. Together, the values of the variables in `VARIABLES` represent the machine's state. So, the initialization defines the initial state containing the initial values of these variables. Again, operations define transitions between two states. Outgoing from the current state, executing an operation modifies the values of the variables to those in the succeeding state. Operations can be modeled with guards by using `SELECT` substitutions. Initialization and operations are composed of substitutions which can be compared to statements in imperative programming languages. In combination, the states and transitions define a transition system. Values for constants are constrained via the `PROPERTIES` clause which is composed of predicates. An invariant is a predicate that must be true in each reachable state of the transition system. Furthermore, other constructs are relevant for verification such as preconditions and assertions. Local variables within an operation and the initialization can be introduced via the `VAR` substitution. With clauses such as `INCLUDES`, `EXTENDS`, `SEES` or `USES`, it is possible to include other B components within a machine.

Below, an example of a lift [3] in B is shown. This example is composed of a variable, an invariant, an initialization, operations, predicates, expressions and substitutions.

```
MACHINE Lift
VARIABLES  floor
INVARIANT  floor : 0..100
INITIALISATION floor := 0
OPERATIONS
    inc = PRE floor<100 THEN floor := floor + 1 END;
    dec = PRE floor>0 THEN floor := floor - 1 END
END
```

Listing 1: Example of a Lift in B

2.2 ProB

PROB is an animator, constraint solver, and model checker for models that are modeled using formal methods. It is implemented in Prolog and supports specification languages such as B, Event-B, Z, XTL, CSP-M, TLA+, and Alloy. The belonging specifications can be animated and verified with PROB. [4]
PROB contains explicit-state model checking [4], LTL model checking [5], and symbolic model checking [6] which can all be used to check these specifications for errors. Symbolic model checking also includes bounded model checking.

XTL specifications can be modeled by PROB if they are implemented containing the predicates `start/1`, `trans/3`, and `prop/2`. This is also a possibility to integrate interpreters in PROB. [4] The predicate `start/1` defines the XTL specification's initial states. In contrast, `trans/3` expects an event and a state and returns the succeeding state after apply-

[3]https://www3.hhu.de/stups/handbook/prob2/prob_handbook.html
[4]https://www3.hhu.de/stups/prob/index.php/Other_languages

ing the given event on the given state. Again, the predicate `prop/2` expects a state and returns the system's properties.

Furthermore, two graphical user interfaces are built on top of PROB: PROB Tcl/Tk[5] and PROB2-UI[6].

2.3 B2Program

B2PROGRAM is a code generator for high-level B specifications that aims code generation for multiple high-level languages. Currently, code generation to Java and C++ are supported. It provides libraries containing the implementation of the B types for both languages. These libraries are then included by the generated code. As B2PROGRAM supports high-level B specifications, the implemented B data types consist of implementations for high-level data types such as sets and relations. The implementation of a B data type contains functions for each operator that can be applied to the corresponding type. E.g. a set is represented by `BSet` which has a function `union` that can be applied to another `BSet` to calculate the union of both sets. In practice, high performance has been achieved by using persistent data structures for the implemented data types. However, since B2PROGRAM uses high-level libraries for high-level B constructs, memory usage cannot be verified. Thus, the code generator cannot be used for embedded systems. Nevertheless, it is applicable for simulation, demonstration, and data validation. [7]

Below, the Java translation of the Lift portrayed in Listing 1 is shown. Here, one can see that the addition uses the function `plus` in the class `BInteger` instead of the Java addition. Furthermore, the generated code gets rid of constructs that are relevant for verification such as the invariant and preconditions.

```
import de.hhu.stups.btypes.BInteger;
import de.hhu.stups.btypes.BUtils;

public class Lift {
    private BInteger floor;

    public Lift() { floor = new BInteger(0); }
    public void inc() { floor = floor.plus(new BInteger(1)); }
    public void dec() { floor = floor.minus(new BInteger(1)); }
}
```

Listing 2: Translation of the Lift in Listing 1

2.4 Lustre Language

LUSTRE is a formal, declarative and synchronous data-flow programming language that was introduced in the 1980s. Its main application fields are avionics, transportation and energy e.g. the flight control of Airbus A320, the emergency shutdown of nuclear plants, and autonomous subway trains are implemented using SCADE which bases on LUSTRE [8]. Some Simulink models are translated to LUSTRE for verification. e.g. the Transport

[5]https://www3.hhu.de/stups/prob/index.php/ProB_Tcl/Tk_Architecture
[6]https://www3.hhu.de/stups/prob/index.php/ProB2_JavaFX_UI

Class Model (TCM) [9] and the Docking Approach of the Space Shuttle to the International Space Station [10]. It is used for programming and verifying reactive systems that are used in safety-critical systems. [11]
Reactive systems are computer systems that react to their environment continuously. The main difference to *interactive systems* is that the latter determine the speed on their own. Instead, the speed of *reactive systems* is given by their environment. [12]
LUSTRE was first introduced with the core of the language and then extended by LUSTRE V4 and finally LUSTRE V6. The LUSTRE core consists of *flows*, *clocks*, *nodes*, *equations*, *assertions* and *expressions*. Expressions are made out of *arithmetic*, *logical*, *comparison* and *temporal* operators and *if-then-else*. LUSTRE V4 extends LUSTRE core by arrays with homomorphism extension. This feature was then replaced by array iterators in LUSTRE V6. Array iterators make use of functional programming concepts to apply operations on arrays. There is also an operator `with` introduced which enables recursion with arrays. Other kinds of recursion are not supported in LUSTRE. Furthermore, LUSTRE V6 contains records, structs and enums that can be used to define user-defined data types. Packages, genericity of nodes and functions are also introduced with LUSTRE V6. The subset languages *ec* and *lic* of LUSTRE are used as an intermediate representation for compilation from LUSTRE V4 and LUSTRE V6 respectively. [13]

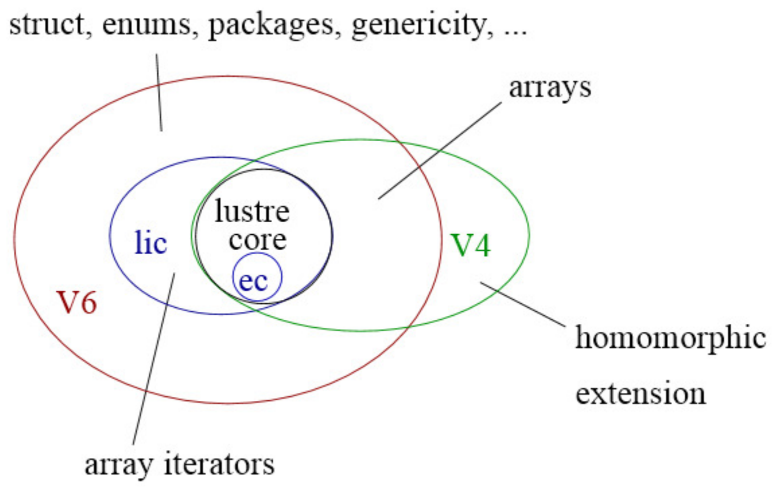

Figure 1: Overview of Lustre Language
[13]

Now the main concepts and language of LUSTRE will be introduced.

2.4.1 Flows and Clocks

Each LUSTRE expression denotes a flow which contains an infinite sequence of values and a clock. The value of an expression at the n-th time step of its clock is the n-th value of the sequence.
A clock is defined by a boolean flow where its time increments when the belonging boolean value is true. The time of a clock is abstract and can thus not be determined

without defining a variable. LUSTRE programs have a cyclic behavior with the fastest clock being the *basic clock* which runs each cycle. It is defined by the boolean flow *true*. Slower clocks are defined by boolean flows whose values are not always true.

The following example shows the cycle of a LUSTRE program, the *basic clock*, and two other clocks *C1* and *C2* with its boolean values. The flow *F1* bases on the *basic clock* and defines the clock *C1*. Again, the flow *F2* bases on the clock *C1* and defines the clock *C2*. Gray-coloured entries denote clock steps where the corresponding flow is not active.

Cycle	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**...**
Flow of Basic Clock	true	true	true	true	true	true	true	...
Time of Basic Clock	1	2	3	4	5	6	7	...
Flow F1 of Clock C1	true	false	false	true	true	false	true	...
Time of Clock C1	1	1	1	2	3	3	4	...
Flow F2 of Clock C2	true	true	true	false	false	false	true	...
Time of Clock C2	1	1	1	1	1	1	2	...

Table 1: Example of Flows and Clocks

2.4.2 Equations

An equation `X=E` defines a variable `X` to an expression `E` according to the *substitution principle*. This means that `X` can be replaced by `E` in any part of the program. Furthermore, an equation describes that a variable `X` behaves like an expression `E`. This is called the *definition principle*. Together, the *substitution principle* and the *definition principle* imply that the order of the equations is not relevant for the program's semantic. A sequence of equations behaves like an equation system as known from mathematics. [14]
As mentioned before, variables and expressions are flows with an infinite sequence of values. So, `X` is the sequence $(X_1, X_2, X_3, \ldots)$ and `E` is the sequence $(E_1, E_2, E_3, \ldots)$. The equation `X=E` defines X_i to be equal to E_i $\forall i \in \mathbb{N}$ as shown in Table 2.
Note that the left-hand side of an equation can also be a tuple of variables if the expression on the right-hand side returns more than one value. This can be the case if there is a tuple of expressions or a node call on the right-hand side.

Cycle	**1**	**2**	**3**	**...**	**N**	**...**
X	X_1	X_2	X_3	...	X_N	...
$Y = X$	$Y_1 = X_1$	$Y_2 = X_2$	$Y_3 = X_3$	...	$Y_N = X_N$	...

Table 2: Variables and Equations

2.4.3 Assertions

Assertions are of the form `assert <boolean expression>`. The given boolean expression which is always sampled on the *basic clock* is an assumption that is expected to be always true. Thus, assertions are used to model the program's environment. It is not used to check whether a property is fulfilled as from imperative programming languages.

Assume a program that models the state of a button with the variables *pressed* and *not_pressed*. Then an assertion could be `assert (pressed and not not_pressed)` `or (not pressed and not_pressed).`

2.4.4 Expressions

LUSTRE contains the basic types boolean, integer, real, and tuples. With LUSTRE V6, it is also possible to define user-defined types. Expressions consist of variables, constants, and node calls of these types and operators that can be applied to them. LUSTRE includes temporal operators (`pre, ->, when, current, merge`) and data operators including arithmetic operators (`+, -, *, /, div, %`), logical operators (`and, or, not, #, nor, =>, xor`), comparison operators (`<, <=, >, >=, =, <>`), and `if-then-else`. [15]

Expressions with Data Operators From the standpoint that each expression is a sequence of values, data operators are applied to the sequences of the operands pointwise as shown in Table 3. Furthermore, the operands of data operations must share the same clock. Otherwise, an error occurs at *clock checking* which is performed similarly to *type checking*. [11]

Cycle	**1**	**2**	**...**	**N**	**...**
C	C_1	C_2	...	C_N	...
X	X_1	X_2	...	X_N	...
Y	Y_1	Y_2	...	Y_N	...
op X	*op* X_1	*op* X_2	...	*op* X_N	...
X *op* Y	X_1 *op* Y_1	X_2 *op* Y_2	...	X_N *op* Y_N	...
if C then X else Y	if C_1 then X_1 else Y_1	if C_2 then X_2 else Y_2	...	if C_N then X_N else Y_N	...

Table 3: Expression Flows of Data Expressions

Expressions with Temporal Operators Temporal operators are `pre`, `->`, `current`, `when`, and `merge`. As `merge` is not supported in this work, it is not described.

The unary operator `pre` references the previous value of its operand. Let $X = (X_1, X_2, X_3, \ldots)$ be an expression's flow, then the flow of $pre(X)$ is equal to $(nil, X_1, X_2, \ldots)$. The undefined value in LUSTRE is represented by `nil`. Both expressions X and $pre(X)$ are sampled on the same clock.
`->` ("Followed-By") is used to define the initial value of an expression. There, the resulting flow accepts the first value of the left-hand side expression as its first value. For all other cycles, it is defined as the value of the right-hand side expression. Let $X = (X_1, X_2, X_3, \ldots)$ and $Y = (Y_1, Y_2, Y_3, \ldots)$ be two expression flows sharing the same clock, then the flow of $X- > Y$ is defined as $(X_1, Y_2, Y_3, \ldots)$. The resulting expression $X- > Y$ also has the same clock as its operands.
The binary operator `when` is used with the left-hand side operand being an expression of any type and the right-hand side operand being a boolean expression. Both operands must also share the same clock. Applying $X\ when\ Y$ results in an expression where X is sampled down to the clock defined by Y. So the expression $X\ when\ Y$ results in the value X_i at the i-th step of its clock if Y_i is true. Otherwise, X is not active but still holds the previous value. Again, `current` is a unary operator that samples the given expression

up to the clock of the boolean expression defining its clock. This allows calculations on expressions with different clocks.
The operand of a `current` expression must not be sampled on the *basic clock*. Let X be an expression with a clock defined by the boolean expression Y, then $current(X)$ takes the value of X the last time Y was true. The resulting clock for $current\ X$ is the clock of Y. [11]

Table 4 shows an example for each of the temporal operators. Gray-coloured entries mark values that are not active in the corresponding cycle.

Cycle	**1**	**2**	**3**	**4**	**5**	**...**
$X = 0 -> 1$	$X_1 = 0$	$X_2 = 1$	$X_3 = 1$	$X_4 = 1$	$X_5 = 1$	...
$Y = pre(X)$	$Y_1 = nil$	$Y_2 = 0$	$Y_3 = 1$	$Y_4 = 1$	$Y_5 = 1$	...
$A = 0 -> pre(A) + 1$	$A_1 = 0$	$A_2 = 1$	$A_3 = 2$	$A_4 = 3$	$A_5 = 4$	...
B	$false$	$true$	$false$	$true$	$false$	...
$C = A\ when\ B$	$C_0 = nil$	$C_1 = 1$	$C_1 = 1$	$C_2 = 3$	$C_2 = 3$	...
$D = current\ C$	$D_1 = nil$	$D_2 = 1$	$D_3 = 1$	$D_4 = 3$	$D_5 = 3$	...

Table 4: Expressions Flows of Temporal Expressions

2.4.5 Nodes

LUSTRE programs can have multiple nodes, with one node being the entry point of the whole program (*main node*). Other *nodes* can then be used as a subprogram in LUSTRE. The header of a *node* declares input parameters, output parameters, and optionally local variables. In contrast, the body contains equations and assertions that must be held by the program. The following example shows a node implementing a simple counter. Starting at 0, the value of `res` increments by `i` each cycle.

```
node Counter (i : int) returns (res : int);
let
    res = 0 -> pre(res) + i;
tel;
```

Listing 3: Simple Counter in LUSTRE as A Node

Furthermore, the input parameters of a node can have different clocks. If the clock of a parameter is not the *basic clock*, then the boolean flow of the corresponding clock must be passed as a parameter and associated with the other parameter. Furthermore, clocks must be declared before the declarations of the variables they are assigned to. Both are illustrated in the example below.

```
node different_clocks(millisecond : bool; a : int when millisecond; b : int)
                        returns (res : int when millisecond);
let
  res = a + b when millisecond;
tel;
```

Listing 4: Example for a Node with Input Parameters Having a Clock Other than the Basic Clock

There is also a graphical modeling language SCADE which bases on LUSTRE. Nodes are visualized as circuits in SCADE. [16]

2.4.6 Verification

A LUSTRE node is usually verified with a corresponding *verification node* checking its behavior. Critical properties are expressed as boolean expressions, conjuncted to a single expression and finally defined as the only output parameter of the *verification node*. This output parameter must then be checked that it is always true for each execution of the program under the assumption of the assertions being true. Possible verification techniques are *model checking* or *proving*. [17]
With the temporal operators in LUSTRE, the language is strong enough to express *safety properties*. *Liveness properties* cannot be covered with the LUSTRE language. Thus, LUSTRE can be seen as a subset of linear temporal logic (LTL). It is more important to check that no dangerous situations occur rather than to check whether an action eventually happens. So *safety properties* are more interesting than *liveness properties* for the verification of LUSTRE programs. [12, 18]

In the following example, a node representing a european lift and its *verification node* are shown. The lowest floor of the lift is the ground floor, while the highest floor of the lift is the 100th floor. It expects a command either to move up, to move down, or to stay on the current floor, and decides whether the command can be executed. The decision and the resulting floor is returned from the node.
A lift should never move up if it is on the highest floor and never move down if it is on the lowest floor despite the given commands to the lift. This is modeled by the variables `exec_up` and `exec_down`. Thus, the variable `floor` only increments if `exec_up` is true and only decrements if `exec_down` is true.

```
node Lift(up, down : bool) returns (exec_up, exec_down : bool; floor : int);
var old_floor : int;
let
    old_floor = 0 -> pre(floor);
    exec_up = up and old_floor < 100;
    exec_down = down and old_floor > 0;
    floor = if exec_up then old_floor + 1 else
                if exec_down old_floor - 1 else old_floor;
tel;
```

Listing 5: Example for a Node Representing a Simple Lift

There is an assumption for this lift. It never expects commands for moving in the first cycle, and the instructions are never both moving up and moving down for the following cycles. Furthermore, the critical properties are:

1. The decision of the lift is never moving up and moving down for the next step. (`only_exec_up_or_down`)
2. The lift never reaches a floor lower than the lowest floor. (`not_lower_than_lowest_floor`)
3. The lift never reaches a floor higher than the highest floor. (`not_higher_than_highest_floor`)

```
node Lift_verif(up, down : bool) returns (ok : bool);
var exec_up, exec_down, only_exec_up_or_down,
    not_lower_than_lowest_floor, not_higher_than_highest_floor : bool;
    floor : int;
let
    exec_up, exec_down, floor = Lift(up,down);

    -- ASSERTIONS
    assert (not up and not down) -> not(up and down);

    -- PROPERTIES
    only_exec_up_or_down = not(exec_up and exec_down);
    not_lower_than_lowest_floor = floor >= 0;
    not_higher_than_highest_floor = floor <= 100;
    ok = only_exec_up_or_down and not_lower_than_lowest_floor and
         not_higher_than_highest_floor;
tel;
```

Listing 6: Example for a Verification Node

3 Overview of Approaches for Simulation and Verification

This section describes both approaches followed in this work to simulate and verify LUSTRE programs in PROB. While the first approach is implementing a LUSTRE AST interpreter in Prolog, the second approach is translating LUSTRE programs to B.
During this work, both approaches and the LUSTRE parser are implemented. Instead of implementing a parser, an existing parser could have been used. However, there are no portable LUSTRE parsers. Furthermore, implementing a parser from the ground up provides more flexibility according to the information stored in the AST nodes and optimization.
The tasks of the parser are composed of lexing, parsing, semantic checking, and preprocessing data for the LUSTRE AST interpreter and LUSTRE2B. First, a LUSTRE program is read from a file as a char stream. After checking it for lexing errors, it is then transformed to a token stream. The resulting token stream is then passed to the parser which checks it for parsing errors. If there are no parsing errors, then the resulting AST is created. Semantic checking is then applied to the AST which also performs some optimizations. On the one hand, it adds semantic information to the AST. On the other hand, the order of equations and assertions is reordered such that sequential interpretation and sequential code generation from the AST are possible. Furthermore, some expressions are rewritten by using the idea of the *substitution principle*. There are also descriptions and temporary variables introduced during the parsing step. Descriptions show the original LUSTRE constructs that belong to node instances and introduced temporary variables. Finally, preprocessing are applied to the AST which generates preprocessed data. During this step, it is determined which variables are stored in the program's state. It also generates the list of clocks and the clock hierarchy. After applying all these steps successfully, an optimized semantic AST is provided together with the preprocessed data to the LUSTRE AST interpreter and LUSTRE2B. As both approaches are implemented using the same abstract syntax tree (AST), the LUSTRE parser is the parser frontend for both approaches. In contrast, the LUSTRE AST interpreter and LUSTRE2B are the two backends as portrayed in Figure 2.

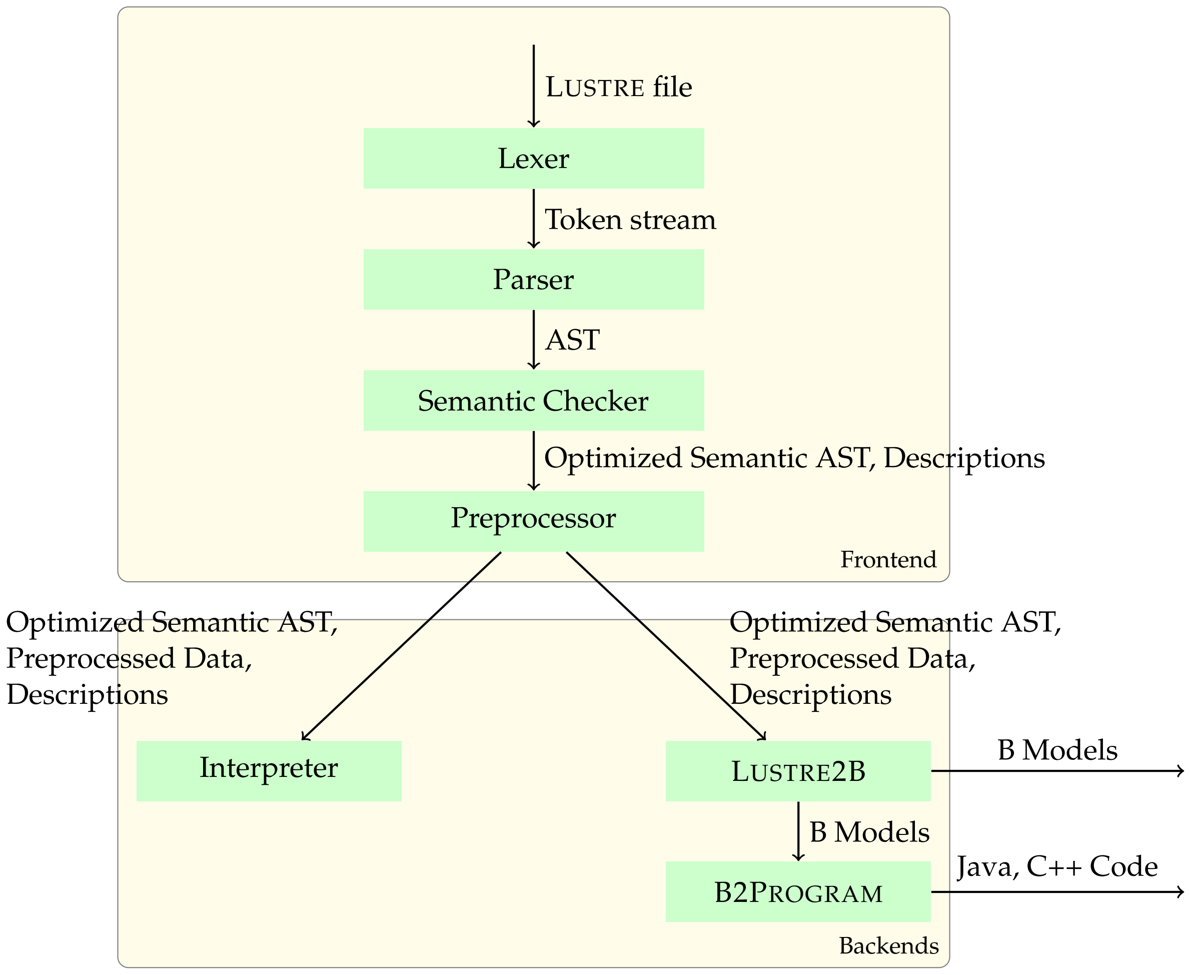
LUSTRE file
Lexer
Token stream
Parser
AST
Semantic Checker
Optimized Semantic AST, Descriptions
Preprocessor
Frontend
Optimized Semantic AST, Preprocessed Data, Descriptions
Optimized Semantic AST, Preprocessed Data, Descriptions
Interpreter
LUSTRE2B
B Models
B Models
B2PROGRAM
Java, C++ Code
Backends

Figure 2: Overview with Frontend and Backends

A LUSTRE AST interpreter could be implemented as an XTL specification in Prolog containing the predicates `start/1`, `trans/3`, and `prop/2`. To achieve a better performance, the interpreter is compiled together with PROB instead. Loading a LUSTRE program requires the implementation of an XTL specification containing the predicates named before which are dispatched on `lustre_start/2`, `lustre_trans/3`, and `lustre_prop/2` in the LUSTRE interpreter respectively. The additional parameter for `lustre_start/2` is explained later in Section 6. This approach enables simulation, explicit-state model checking, and LTL model checking of LUSTRE programs in PROB. Nevertheless, some features of PROB cannot be supported by this approach e.g. symbolic model checking. But again, it provides the opportunity to support LUSTRE programs with real numbers using floating-point numbers in Prolog.
In contrast, the other approach is translating LUSTRE programs to B machines and then simulating and verifying them in PROB. This approach also provides the opportunity to apply B2PROGRAM on the translated B models to generate Java and C++ code for simulation. In contrast to the LUSTRE AST interpreter, symbolic model checking is supported by following this approach. But as B does not support floating-point numbers, LUSTRE

programs with real numbers cannot be translated. Translating LUSTRE to B also enables applying other B-method tools such as the proving tool ATELIERB[19].
There is one disadvantage that comes with both approaches. As real numbers might lead to a large state space, verification via symbolic model checking might be more suitable for LUSTRE programs with real numbers. But symbolic model checking is only supported for B which again does not support real numbers.

Interpretation of LUSTRE programs in Prolog is described in Section 6. Translation of LUSTRE programs to B machines is explained in Section 7. Furthermore, the performance of simulation and verification is later analyzed for both approaches (see Section 8).

4 Approaches for Implementation of the Parser

In this section, different approaches for implementing the parser are described and compared with each other. One approach is implementing the parser in Prolog directly, while the other approach is implementing the parser in Java using a parser generator such as SABLECC[7] or ANTLR[8]. During this section, the advantages and disadvantages of these approaches are outlined and evaluated. Finally, the decision is made to implement the LUSTRE parser in Prolog.

4.1 Implementing the Parser in SableCC

SABLECC is a parser generator that is implemented in Java. When using SABLECC, one needs to write regular expressions for tokens and an LALR(1) grammar for parsing. It is also necessary to implement the mapping of the concrete syntax tree (CST) to the abstract syntax tree (AST) in the grammar file. SABLECC then generates deterministic finite automata for the tokens, and the parser. Furthermore, SABLECC creates the AST, and tree walker classes for the visitor interface. So, the main advantage is that the parser is generated from the grammar directly. [20]
As the grammar can be written down as an LALR(1) grammar, it is not necessary to left-factorize or eliminate left recursion in the grammar. On the one hand, error messages for lexing and parsing errors are also already implemented in the generated parser. On the other hand, these error messages are hard to understand and cannot be extended easily. As the mapping of the CST to the AST must be implemented in the grammar file directly, SABLECC is not very flexible. Although one can add information e.g. semantic information to the AST nodes, it is still not maintainable. This could lead to the problem, that a node must be reimplemented when the grammar file changes.
There are no disadvantages according to the implementation of LUSTRE2B. But as the interpreter must be implemented in Prolog to integrate it in PROB, it would be necessary to generate a Prolog representation of the AST when using SABLECC as a parser generator. Furthermore, it would be required to implement an interface between Java and Prolog.

[7]http://sablecc.org/
[8]https://www.antlr.org/

4.2 Implementing the Parser in ANTLR

ANTLR is a parser generator for LL(K) grammars that is also implemented in Java. One also needs to write regular expressions for tokens and the grammar for parsing. In general, left-recursion is not allowed in LL(K)-grammars. The implementation of ANTLR handles left-recursion by transforming the grammar before generating the parser. So, it is not necessary to left-factorize and eliminate left recursion when writing the grammar. [21]
Like SABLECC, an advantage is that the parser is generated from the grammar directly. In comparison to SABLECC, it turns out to be more maintainable. In contrast, there is no need to implement the mapping of the CST to the AST in the grammar file. So, the parser generator generates the CST only. Similar to SABLECC, it also generates the lexer, the parser, and the tree walker classes. As one can implement the mapping of the CST to the AST manually, it is more flexible. This also provides the opportunity to add semantic information to the AST nodes in a maintainable way.
Error messages for lexing and parsing errors in ANTLR are better than those in SABLECC. Nevertheless, it is still hard to improve the error messages for lexing and parsing errors.
There is also the disadvantage that it would require generating a Prolog AST representation as the interpreter must be implemented in Prolog. Similar to SABLECC, it would also be necessary to implement an interface between Java and Prolog.

4.3 Implementing the Parser in Prolog

Another possibility is implementing the parser in Prolog directly. As Prolog supports definite clause grammars (DCGs), the grammar for the parser can also be written down. The input can be accessed efficiently because DCGs are implemented using difference lists. But, there is a disadvantage that Prolog only supports LL(1) grammars. Thus, the grammar has to be left-factorized and left recursion must be eliminated. This leads to a slightly more complicated grammar. Furthermore, the grammar is right-associative after eliminating left recursion. Therefore, expression AST nodes have to be transformed such that the left-associativity of binary operators is ensured. [22]
With Prolog, it is even possible to generate the belonging AST node with fields for semantic information directly. While tokens can be implemented as regular expressions in ANTLR and SABLECC, one needs to write a grammar for each token in Prolog. This is due to the reason that the operators for regular expressions (*alternate*, *concatenation*, *closure*) are supported in ANTLR and SABLECC, but not in Prolog.
Lexing and parsing errors are detected by the DCG, but there are no error messages returned. On the one hand, implementing the parser in Prolog has the disadvantage that lexing and parsing error messages must be implemented. On the other hand, the error messages can be implemented such that they are more flexible and informative than the default messages in ANTLR and SABLECC.
As recursive programming is fundamental in Prolog, it is also easier to implement the visitors in Prolog compared to SABLECC and ANTLR. A big advantage of implementing the parser in Prolog directly is the fact that the generated AST can be passed to the interpreter directly.

4.4 Conclusion

All in all, implementing the parser in Prolog directly turns out to be the best solution. Implementing parsing and lexing is slightly easier when using ANTLR or SABLECC rather than Prolog. For the generation of the AST with semantic information, ANTLR and Prolog seem to be better than SABLECC. In contrast, error messages are more flexible when implementing the parser in Prolog rather than in ANTLR or SABLECC. But two big advantages lead to the decision of implementing the parser in Prolog. The first advantage is the fact that handling the AST in Prolog is easier than in Java. One main goal of the parser is simplifying both, interpretation in Prolog and translation to B. As explained later in Section 5, the tasks of this parser are more complex. The second advantage is that the generated AST can be passed to the interpreter directly. In ANTLR and SABLECC, it would be necessary to generate the Prolog AST representation.

5 Parser

This section describes the structure of the parser containing the steps of lexing, parsing, semantic checking, and preprocessing for the backends. The tokens and the AST will be described in detail. Furthermore, it will be explained how the steps of semantic checkings (scoping, type checking, clock checking, recursion checking, definition checking, and cycle checking of equations [11]) are implemented. Some optimizations are applied on the AST to ease interpretation in Prolog and translation to B. It also generates preprocessed data and descriptions that are provided to both backends. LUSTRE constructs that are not supported in this work yet are portrayed in Appendix B.

5.1 Lexing

The lexer expects a LUSTRE file as an input stream and divides it into tokens. A lexing error occurs if there are no matched tokens for parts of the LUSTRE program during the tokenizing process. Each token is associated with different groups as shown in Table 5.
As Prolog does not support regular expressions directly, the behavior of its used operators (*alternative, concatenation, repetition*) is implemented for each token individually. So, each token is realized using a regular grammar.

Furthermore, a token is represented by a compound term storing the corresponding position. In addition to the position information, the tokens for identifiers and literals also store a field for the identifier and the value respectively. A position is of the form `pos(Line, Column, Length)`.
E.g. the token `node` is represented as `node(Pos)`, while the token `identifier` is represented as `identifier(Pos, ID)`.

All tokens' representations and regular expressions are shown in Appendix A.

Token Group	**Tokens**
Keywords and Operators	`node`, `var`, `let`, `tel`, `returns`, `int`, `bool`, `real`, `if`, `then`, `else`, `assert`, `+`, `-`, `*`, `/`, `%`, `mod`, `<`, `<=`, `=`, `<>`, `>`, `>=`, `and`, `or`, `not`, `pre`, `->`, `when`, `current`, `^`, `..`
Identifier and Literals	Identifier, Integer Literal, Real Literal, `true`, `false`, `nil`
Separators	`,`, `:`, `;`, `(`, `)`, `[`, `]`
Others	Whitespace, Comment

Table 5: Overview of Tokens

Consider the following example:

```
node Simple(x : int) returns (res : int);
let
    res = x;
tel;
```

Listing 7: Simple Lustre Node

The resulting token stream is a list containing the following tokens:

```
node: POSITION: Line 1, Column 1, Length 4
identifier: POSITION: Line 1, Column 6, Length 6; NAME: Simple
lpar: POSITION: Line 1, Column 12, Length 1
identifier: POSITION: Line 1, Column 13, Length 1; NAME: x
colon: POSITION: Line 1, Column 15, Length 1
int: POSITION: Line 1, Column 17, Length 3
rpar: POSITION: Line 1, Column 20, Length 1
returns: POSITION: Line 1, Column 22, Length 7
lpar: POSITION: Line 1, Column 30, Length 1
identifier: POSITION: Line 1, Column 31, Length 3; NAME: res
colon: POSITION: Line 1, Column 35, Length 1
int: POSITION: Line 1, Column 37, Length 3
rpar: POSITION: Line 1, Column 40, Length 1
semicolon: POSITION: Line 1, Column 41, Length 1
let: POSITION: Line 2, Column 1, Length 3
identifier: POSITION: Line 3, Column 5, Length 3; NAME: res
equal: POSITION: Line 3, Column 9, Length 1
identifier: POSITION: Line 3, Column 11, Length 1; NAME: x
semicolon: POSITION: Line 3, Column 12, Length 1
tel: POSITION: Line 4, Column 1, Length 3
semicolon: POSITION: Line 4, Column 4, Length 1
```

Listing 8: Example of Token Stream for Listing 7

5.2 Parsing

This subsection describes the implementation of the parsing step and the representation of the AST. Despite arrays, these are also the AST nodes for all LUSTRE constructs that are part of the supported subset of LUSTRE in this work. LUSTRE constructs that are not supported yet are portrayed in Appendix B.

After lexing is applied to the LUSTRE program successfully, the resulting token stream is passed to the parser. It is then checked for parsing errors. Once the LUSTRE program is parsed successfully, the resulting Prolog AST is created as output. Each AST node is generated from the corresponding applied nonterminal.

The context-free grammar of the supported subset of LUSTRE is taken from [15]. For the implementation, it is left-factorized and containing left recursions are eliminated. This is necessary because Prolog DCG parsers are LL(1) parsers. Furthermore, the grammar is right-associative after eliminating left recursion. Therefore, expression AST nodes have to be transformed such that the left-associativity of binary operators is ensured.

The AST of the whole LUSTRE program is represented by a compound term with `program` as a functor and a list of nodes as arguments. A node is represented by a compound term with `node` as a functor wrapping its position, its name, and its children AST nodes. These are input parameters, output parameters, local variables, and the node's body. Input parameters, output parameters, and local variables are all represented as a list of declarations (see Table 7). In contrast, the node's body is an AST node that stores a list of equations and assertions within a functor `body` (see Table 8).

Program and Node AST Nodes	**Representation**
Program	program(Nodes)
Node	node(Pos, NodeName, Parameters, Outputs, Locals, Body)

Table 6: Overview of Program and Node AST Nodes

A declaration AST node contains a list of identifiers, a type, and optionally a clock. All variables within a declaration are associated with the same type and clock. This LUSTRE parser supports primitive types (boolean, integer, real) and array types. The compound term for primitive types stores the position and an atom which is either `boolean`, `integer` or `real`. Additionally, the one for array types stores an expression defining its size. Note that the work for array types has not been finished. It is only supported in the lexer and parser. While the *basic clock* is represented by the atom `$basic`, other clocks are represented by a compound term wrapping the clock's name. There are no special reasons not to wrap `$basic` within the functor `clock`. Identifiers cannot begin with a *dollar* in LUSTRE. To avoid collision with the allowed clock name `basic`, the internal representation of the *basic clock* starts with a *dollar*.

Declaration AST Nodes	**Representation**
Variable Declarations	declarations(Pos, Identifiers, Type, Clock)
Primitive Type	primitive_type(Pos, Type)
Array Type	array_type(Pos, Type, Expression)
Clock	clock(Pos, Identifier)

Table 7: Overview of Declaration AST Nodes

As mentioned before, the AST node for the node body wraps a list of equations and assertions. Again, an equation stores a list of identifiers (representing the left-hand side) and an expression (representing) the right-hand side). In contrast, an assertion stores an expression only.

Body AST Nodes	**Representation**
Node Body	body(EquationsAndAssertions)
Equation	equation(Pos, Identifiers, Expression)
Assertion	assertion(Pos, Expression)

Table 8: Overview of Body AST Nodes

Each expression node has a field storing its position and a field storing the semantic information that is composed of type and clock information. It is of the form `[type/Type, clock/Clock]`. Before type checking and clock checking are applied to the AST, the type and clock are `untyped` and `unclocked` respectively. After type checking and clock checking, `untyped` and `unclocked` are replaced by the respective type and clock.
The AST nodes for boolean, integer, and real literals store the respective value. In contrast, an identifier AST node stores the corresponding identifier. Other expression AST nodes store the respective operands as children nodes which are also expressions. Additionally, binary and unary expressions contain the corresponding operator. While, unary operators are represented as `uminus`, `not`, `pre`, and `current` respectively, the binary operators are represented as `plus`, `minus`, `mul`, `div`[9], `mod`, `and`, `or`, `less`, `lessequal`, `greater`, `greaterequal`, `equal`, `unequal`, `fby`[10], and `when` respectively.
Node calls contain a node identifier and parameters as children nodes. The main difference between an identifier expression and a node identifier is that the node identifier has a field storing the input and output variables' name, and the *node ID*. This field is of the form `[input/InputParameters, output/OutputParameters, id/ID]`. `InputParameters` and `OutputParameters` initially store the value `empty`, while `ID` is initially assigned to `none`. All of them are replaced by the respective data at semantic checking. Even though this implementation makes the AST more complicated, required data for both backends can be accessed easily. On the one hand, one does not need to access the node parameters through the scope table of nodes. On the other hand, the node call's instance can be accessed directly. Note that each node call simulates an individual node instance. The list of parameters is a list of expressions and therefore wrapped within the functor `expression_list`. Expression lists can be used in the context of both within and outside a node call. As later explained in Section 5.3.5, expression lists are rewritten to simplify interpretation and translation to B. There, expression lists within and outside a node call are treated similarly. That is why the representation of the node call AST does not get rid of the functor `expression_list`.

Expression AST Nodes	**Representation**
Boolean Literal	boolean(Pos, Value, SemanticInformation)
Integer Literal	integer_val(Pos, Value, SemanticInformation)
Real Literal	real_val(Pos, Value, SemanticInformation)
Identifier	identifier(Pos, Name, SemanticInformation)
Unary Operation	unary(Pos, Operation, Expression, SemanticInformation)
Binary Operation	binary(Pos, Operation, Lhs, Rhs, SemanticInformation)
If-Then-Else	if_then_else(Pos, Condition, Then, Else, SemanticInformation)
Node Call	node_call(Pos, NodeIdentifier, Parameters, SemanticInformation)
Node Identifier	node_identifier(Pos, NodeName, NodeInformation, SemanticInformation)
Expression List	expression_list(Pos, Expressions)
Array Enumeration	array(Pos, Entries, SemanticInformation)
Array Element Access	array_access(Pos, Array, Access, SemanticInformation)
Array Index Access	index(Pos, Index)
Array Slice Access	slice(Pos, Start, End)

Table 9: Overview of Expression AST Nodes

The array enumeration's AST node consists of the functor `array` and a list of expressions. As the expressions are the entries of an array, they are not handled like expres-

[9]Note that this is the operator /, not div
[10]Note that this is the followed-by operator ->, not fby

sion lists which are used as an expression. Thus, they are not wrapped in the functor `expression_list`. Elements in an array can be accessed either via indexing or slicing.

As an example the resulting AST node for `y = x + 1` starting at line 1 and column 2 is as follows:

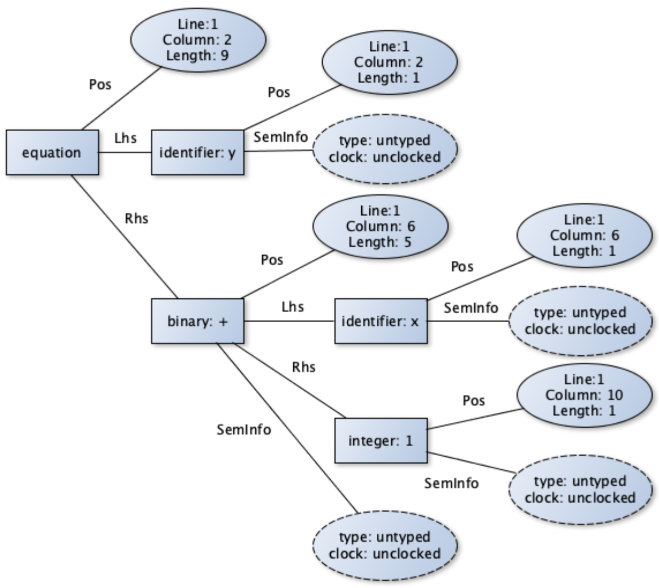

Figure 3: AST Node for `y = x + 1`

As mentioned before, type and clock information is added at type and clock checking respectively. Thus, the type and clock of each expression AST node are `untyped` and `unclocked` after parsing.

5.3 Semantic Checking

Semantic checking includes scoping, type checking, clock checking, recursion checking, definition checking, and cycle checking of equations [11]. Scoping ensures that all used variables and nodes are defined. It generates the scope tables for variables, and the scope table for nodes which are all used by type checking and clock checking. Furthermore, it adds the input and output parameters' names to node identifiers. In contrast, type and clock checking add semantic information to the AST and check it for semantic errors. Checking for recursions is necessary as each invoked node is a subprogram of the invoking node. Allowing recursions would lead to an infinitely large program. Each output variable and local variable must be defined in exactly one equation. Furthermore, input parameters cannot be redefined. This is checked by definition checking. Both recursion checking and definition checking do not modify the AST. A cyclic dependency between variables in an equation system might lead to non-deterministic behavior. As LUSTRE does not allow non-determinism [15], it must be checked that there are no cyclic dependencies. If cycle checking of equations succeeds, then the order of equations and assertions in the AST has changed such that sequential interpretation and sequential code generation is possible. Cycle checking also rewrites some expressions which eases cycle checking on the one hand, and also interpretation and translation to B on the other hand. As a result of semantic checking, an optimized semantic AST and some descriptions are generated. The workflow of the semantic checker is portrayed in Figure 4.

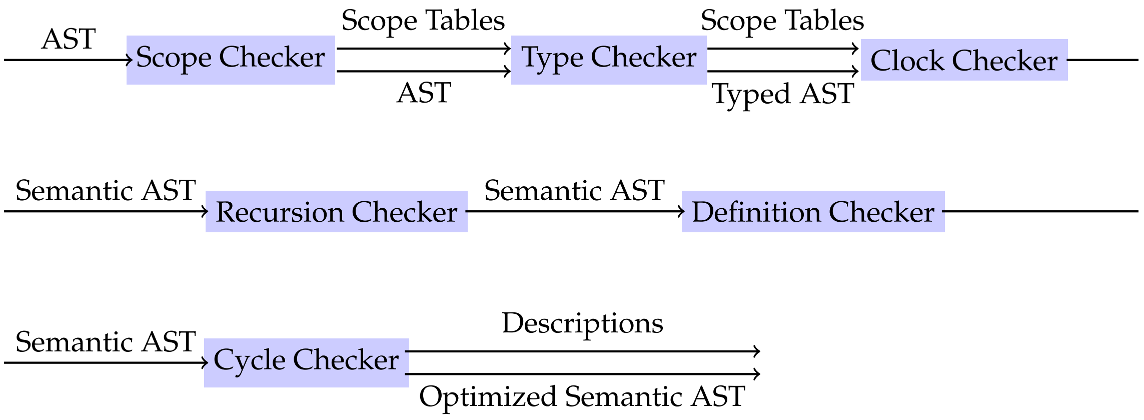

Figure 4: Semantic Checker of Parser Frontend

All steps are implemented using the visitor pattern [23] and will be described more in detail now.

5.3.1 Scoping

Scoping is separated into two steps before scope checking is applied. These steps are global and local scoping. In the supported subset of LUSTRE, all variables are declared within a node only. There are only declarations of nodes outside a node. So, global scoping generates a scope table for nodes, while local scoping generates scope tables with variables for each node. After scoping is applied, the generated scope tables are used for scope checking. It must be ensured that all used variables and nodes are declared exactly once. As explained before, the scope tables are also passed to the type checker and clock checker as they are implemented storing semantic information.
The node scope table stores key-value pairs, where a node's name is mapped to its signature. Again, the signature of a node is composed of the identifiers, the types and the clocks of the input and output parameters. As the node scope table stores semantic information of all nodes, they are used for semantic checking of node calls. Because clocks can be passed to node calls, clock checking also makes use of the invoked node's parameter names. This will be explained in Section 5.3.2. It is also the reason why the AST node of each node call stores the input and output identifiers of the invoked node as portrayed in Section 5.2. Initially, the corresponding fields store the value `empty` which is replaced by the respective list of identifier names at this step.

Consider a LUSTRE program containing the nodes in Listing 5 and Listing 6. The resulting node scope table is shown in Table 10.
Nodes consist of variables that are declared as input parameters, output parameters, and local variables. The variable scope table of a node also consists of key-value pairs where

each variable name is mapped to its type and clock. The resulting variable scope table for the node in Listing 5 is shown in Table 11.

Node	**Signature**
Lift	up : boolean/$basic, down : boolean/$basic → exec_up : boolean/$basic, exec_down : boolean/$basic, floor : integer/$basic
Lift_verif	up : boolean/$basic, down : boolean/$basic → ok : boolean/$basic

Table 10: Node Scope Table of Program containing Listing 5 and Listing 6

Variable	**Type**	**Clock**
up	boolean	$basic
down	boolean	$basic
exec_up	boolean	$basic
exec_down	boolean	$basic
old_floor	integer	$basic
floor	integer	$basic

Table 11: Scope Table of Listing 5

All scope tables are represented as AVL trees which are supported in Prolog. Instead of AVL trees, it would also be possible to use lists. But it has been underlined that the operations on AVL trees e.g. insert, change, delete have a better performance than those on lists especially for a large amount of data [24].
After scoping and scope checking are completed for the whole program, the scope tables and resulting AST are passed to the type checker and clock checker.

5.3.2 Type Checking and Clock Checking

During this step, the AST is checked for type and clock errors. Furthermore, semantic information is added to each expression AST node, if type checking and clock checking succeed as shown in Figure 5.
As LUSTRE is a strongly typed language, this step can be applied at compile-time [25]. Clock checking is equivalent to type checking and is also applied to a *type system* which is called the *clock calculus* [26].
As mentioned before, both steps are implemented using the resulting AST and scope tables. With these scope tables, it is possible to look up the type and clock of all variables and nodes.

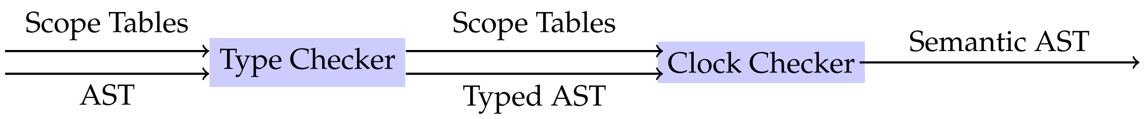

Figure 5: Approach with Generating Semantic AST

Instead of generating a semantic AST, it would also be possible to check the AST for semantic errors without adding any information. The scope tables would then be passed to the backends to access semantic information of variables and nodes. This would lead to a better performance at parsing because type and clock checking would be performed using the AST and the scope tables without modifying the given AST as shown in Figure 6.

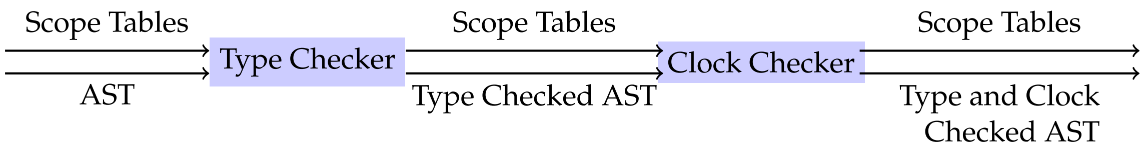

Figure 6: Approach without Generating Semantic AST

Semantic information is required to interpret the LUSTRE AST. Precisely, the clock must be determined at the interpretation of an equation. Consider an equation `x = E` whereas `x` is sampled on a clock `C`. Then the effect of this equation only applies if the clock `C` is active.
Without a semantic AST, it would be necessary to look up the clock of variables that are used on the left-hand side of an equation. In contrast, a semantic AST provides the opportunity to access semantic information and thus access the clock directly. Note that the lack of performance when filling the AST with semantic information only occurs once. While the approach without a semantic AST would lead to worse performance at interpretation, a semantic AST makes it possible to look up the clock more efficiently. That is why creating an AST with semantic information is better for interpretation.
Concerning LUSTRE2B, an AST with semantic information would also simplify the generation of B code. Both type and clock information are required when translating LUSTRE to B. On the one hand, they can be extracted from the expression nodes directly. On the other hand, they do not need to be recalculated from the scope tables either. All in all, generating a semantic AST simplifies both interpretation in PROB and translation to B.
Under the assumption that the variables `x` and `y` are declared as integer variables that are sampled on the *basic clock*, the resulting semantic AST node for the example in Figure 3 is as follows:

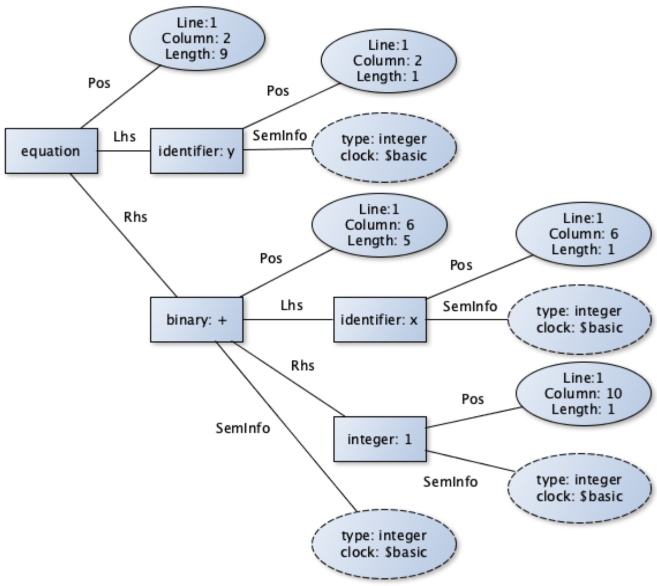

Figure 7: Semantic AST Node for `y = x + 1`

Type checking and clock checking are straightforward except for clock checking of node invocations. LUSTRE operations never cast an expression's type implicitly. So, type casting must always be specified in a LUSTRE program explicitly. This is not part of the supported subset of LUSTRE yet. There are two difficulties for clock checking of node invocations: On the one hand, the *basic clock* of an invoked node must not be the *basic clock* of the whole program. On the other hand, clocks can be passed as parameters through a node. Thus, the identifier of a clock that is passed through the node call can differ from the corresponding identifier at the node's declaration. [11, 14] Note that each clock in the supported subset of LUSTRE is defined as a boolean variable. This is also the reason why the node scope table also contains the complete signature of each node including the identifiers of the input and output parameters. So, for clock checking of node calls it is necessary to apply the following steps:

1. Rename the *basic clock* in the input and output parameters of the invoked node to the clock passed by the invoking node
2. Rename clock names in the node declaration to the corresponding clock names that are passed by the invoking node

Thus, the clocks in node calls are renamed to those clocks in the context of the invoking node. Note that LUSTRE's semantics enforces the clocks which are passed to a node call to be visible from outside the node [11]. This means that the clocks of input and output parameters must be passed as parameters to the node calls explicitly. Furthermore, expressions using these clocks must be located within the invoked node.

The following example in Listing 9 is taken from [15]. It shows a node call where the parameters are sampled to the clock `second`. So the input and output parameters of the node call are sampled on `second`, while those of the node `SLOW_STABLE` are sampled on the *basic clock*. This means that the *basic clock* of the invoked node `SLOW_STABLE` in `SLOW_TIME_STABLE` is `second`.

As result, the declaration of `SLOW_STABLE` defines its clock to be `($basic, $basic)` $\rightarrow$ `$basic`, while it is used as `(second, second)` $\rightarrow$ `second` in the context of `SLOW_TIME_STABLE`.

Thus, it is necessary to rename the *basic clocks* in the node call of `SLOW_STABLE` to `second`.

```
node SLOW_TIME_STABLE(set, second : bool; delay : int) returns (level : bool when second);
let
  level = SLOW_STABLE((set,delay) when second);
tel;

node SLOW_STABLE(set : bool; delay : int) returns (level : bool);
let ... tel;
```

Listing 9: Node Invocation where Basic Clock of Invoked Node Must Be Renamed

Another example shows a node `nclock2` which expects the clock `clock1` as a parameter. It is also the clock of the return parameter. While this clock is named `clock1` in the context of the node `nclock2`, it is named `y` in the context of the invoking node `nclock1`.

So, the return parameter of the node call is sampled on `y` in the context of `nclock1`, while the declaration of `nclock2` declares the return clock as `clock1`. Similar to renaming the *basic clocks* in the input and output parameters, `clock1` is renamed to `y` for the node call.

```
node nclock1(x,y : bool) returns (res1 : bool when y);
let
  res1 = nclock2(x,y);
tel;

node nclock2(val, clock1 : bool) returns (res1 : bool when clock1);
let
    res1 = val when clock1;
tel;
```

Listing 10: Node Invocation where Clocks Passed as Parameters Must Be Renamed

5.3.3 Checking Recursions

Invoked nodes within expressions are subprograms of the invoking node. It means that these nodes are connected to the invoking node in a circuit. So, the parameters that are passed to the invoked nodes are the connection between the two nodes. Because invoked nodes are subprograms, recursions would lead to an infinitely big circuit representing the program. LUSTRE supports static recursion using arrays in combination with the `with` operator only. This feature is not supported in the subset of LUSTRE in this work. Other kinds of recursions are not allowed in the language of LUSTRE and must be detected at parsing. [11]
To achieve this, the dependencies between nodes must be detected. *A node M contains the node N as a subprogram if M invokes N or there exists another node L such that M invokes L and L contains N as a subprogram.* First, all direct connections between two nodes in the whole

program are inspected. Afterwards, *transitive closure* is applied to the result to create the dependency graph. While the nodes in the dependency graph represent the nodes in the program, the edges are directed and describe the relation *contains as a subprogram*.
With this dependency graph, the program can be checked for recursions. If there exists a node such that it contains itself as a subprogram, then a recursion is detected.

The following example shows two nodes `Recursion1` and `Recursion2` that invoke each other which results in the graph in Figure 8. Applying *transitive closure* on this graph results in the graph describing the *contains as subprogram* relation between nodes (see Figure 9). There, one can see two edges that connect a node with itself. So, a recursion is detected in the given program and thus an error is detected.

```
node Recursion1(x : int) returns (res : int);
let
  res = if x = 0 then x else Recursion2(x-1) + x;
tel;

node Recursion2(x : int) returns (res : int);
let
  res = if x = 0 then x else Recursion1(x-1) + x;
tel;
```

Listing 11: LUSTRE Nodes containing A Recursion via Many Node Calls

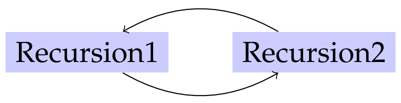

Figure 8: Node Invocation Relation between Nodes in Listing 11

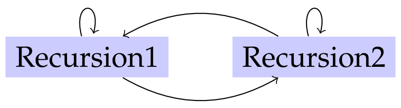

Figure 9: Contains as Subprogram Relation between Nodes in Listing 11

5.3.4 Checking Definitions

During this step, it must be checked that each variable is defined exactly once. This means that each output variable and local variable is defined in exactly one equation, while input variables are not allowed to be used on the left-hand side of an equation. [11]
Note that this step differs from checking variables for redeclarations which is already done in Section 5.3.1.

The following example shows a node that contains all kind of definition errors. While the output variable `res1` appears twice on the left-hand side of an equation, the parameter variable `x` is defined in an equation. Furthermore, one can see that the variables `loc` and `res2` are not defined.

```
node def_errors(x : int) returns (res1, res2 : int);
var loc : int;
let
  res1 = x;
  res1 = 3; -- definition error of output variable
  x = 4; -- definition error of parameter variable
  -- res2 and loc are not defined
tel;
```

Listing 12: LUSTRE Program with Definition Errors

5.3.5 Cycle Checking and Rewriting Equations

After applying all previous semantic checks successfully, the resulting AST is passed to the cycle checker to be checked for cyclic dependencies between variables. First, all node calls in the program are visited to store its *ID* in the AST node. They are enumerated in the order they appear in the original AST. This is implemented by an individual visitor which yields the resulting AST and description for node calls. Before the node body is checked for cyclic dependencies, expression lists, temporal expressions, and if-then-else expressions are rewritten. There are also temporary variables introduced and some equations rewritten. This makes it possible to reduce checking cyclic dependencies within an equation to cycle checking of equations in general.

Rewriting Equations with Expression Lists Equations contain a list of identifiers on the left-hand side and possibly expression lists on the right-hand side. These expression lists might again contain node calls. Furthermore, nodes could be defined with multiple output parameters and thus could return multiple values.

The operators `pre`, `->`, and `current` can be used with expression lists as operands. Again, the left-hand side of `when` might contain an expression list as well. This is also allowed for `then` and `else` expressions of `if-then-else`. All expression lists on the right-hand side of equations are rewritten such that they are top-level as shown in Listing 13.
It could be the case that operands of these operators are node calls that returns more than one value. To rewrite the expression list, it is necessary to store the node calls in temporary variables. This is explained later in this section.

```
-- pre
pre((expr1, ..., exprn)) ⇝ (pre(expr1), ... pre(exprn))

-- ->
(l_expr1, ..., l_exprn) -> (r_expr1, ..., r_exprn) ⇝
(l_expr1 -> r_expr1), ..., (l_exprn -> r_exprn)

-- when
(expr1, ..., exprn) when clock ⇝ (expr1 when clock, ..., exprn when clock)

-- current
current((expr1, ..., exprn)) ⇝ (current(expr1), ..., current(exprn))

-- if-then-else
if cond then (t_expr1, ..., t_exprn) else (e_expr1, ..., e_exprn) ⇝
(if cond then t_expr1 else e_expr1, ..., if cond then t_exprn else e_exprn)
```

Listing 13: Rewriting of `Pre`, `->`, `When`, `Current`, and `If-Then-Else` Expressions with Expression Lists

An equation is rewritten to many equations such that the left-hand side of each equation contains as least variables as possible. This simplifies checking cyclic dependencies between variables.
As a result, there are only more than one variable on the left-hand side of a rewritten equation, if the right-hand side of this equation contains a node call as top-level that returns more than one value. The algorithm is shown below. Note that the portrayed algorithm is imperative, while the implementation is declarative. It is only intended to describe how the idea of the algorithm works. Furthermore, note that all functions are applied without side effects i.e. without modifying the value of its operands. While `head` returns the first element of a list, `tail` returns the sublist without the first element. `lhs` and `rhs` denote functions accessing the left-hand side and right-hand side of an equation respectively. `take_first` returns the first elements of a list, while `remove_first` returns the sublist without the first elements.

Algorithm 1: Rewriting Equation with Expression Lists

```
Input: Equation $Eq$: $Lhs_1$, ..., $Lhs_N$ = $Rhs_1$, ..., $Rhs_M$
Output: List of Equations $ResEqs$: $ResEq_1$; ...; $ResEq_P$;

Ls := lhs(Eq);
Rs := rhs(Eq);
ResEqs := [] ;
while Rs is not empty do
    NextRs := head(Rs);
    if NextRs is node call then
        L := length(output(NextRs));
    else
        L := 1;
    NextLs := take_first(L, Ls);
    ResEqs := append(ResEqs, [NextLs = NextRs]);
    Ls := remove_first(L, Ls);
    Rs := tail(Rs);
return ResEqs;
```

Listing 14 shows a LUSTRE program with a node that has an equation with many variables on the left-hand side and an expression list on the right-hand side. The first expression within the expression list is a node call that returns two values, while the second expression is a variable only. So, this equation is rewritten to two equations as shown Listing 15.

```
node node1(x : int) returns (a,b,c : int);
let
    a,b,c = node2(x), a;
tel;

node node2(x:int) returns(s,t : int);
let ... tel;
```

Listing 14: Example with a Node containing an Equation with Many Variables on Left-Hand Side

```
node node1(x : int) returns (a,b,c : int);
let
    a,b = node2(x);
    c = a;
tel;

node node2(x:int) returns (s,t : int);
let ... tel;
```

Listing 15: Rewritten Equations from an Equation with Many Variables on Left-Hand Side

Instead of rewriting an equation with expression lists on the right-hand side, it would also be possible to rewrite the order of the variables and expressions within the equation on both sides.
But following this approach provides the opportunity to generalize the problem of checking cyclic dependencies within an equation to the problem of checking cyclic dependencies between many equations.

Rewriting Temporal Expressions A big issue when interpreting LUSTRE programs is the evaluation of expressions with temporal operators. On the one hand, expressions can be evaluated according to a specific clock. On the other hand, `pre` can be used to access previous values of an expression flow.

The initial idea was to store current and previous values of all variables according to each clock. How many previous steps were saved for a variable depended on the maximum nesting depth of `pre` operators in which the variable is located in the whole node. When evaluating a variable in an expression, it was necessary to determine its clock and the nesting depth of `pre` operators of the corresponding expression. The variable was then evaluated by looking up its value in the clock's state if the nesting depth is equal to 0. To evaluate previous values, it was necessary to store values of a variable more than one time step previously. Evaluating previous values of a variable was then done by looking up the value of the pair `(NestingDepth, Variable)` in the clock's history. It was also necessary to store the time of each clock to evaluate nested `pre` expressions that are combined with the -> operator.
Nevertheless, it was not possible to cover and evaluate every combination of temporal operators by following this approach. The main problem is caused by the operator `pre` accessing previous values, and the operators `when` and `current` sampling an expression

on a specific clock. In contrast to variables, expressions were not stored in the LUSTRE state and thus cannot be used for the calculation of the next cycle.

Consider the following node and its expression flows. The gray-coloured entries in Table 12 denote values that are not active in the corresponding cycle.

```
node n(B : bool) returns (res1, res2 : int);
var X : int;
let
  X = 1 -> pre(X) + 1;
  res1 = current(pre(X) when B);
  res2 = current(pre(X when B));
tel;
```

Listing 16: Example for Node Describing Evaluation Problem of Temporal Expressions

Cycle	**1**	**2**	**3**	**4**	**5**	**...**
`X`	1	2	3	4	5	...
`B`	true	false	true	false	true	...
`pre(X)`	nil	1	2	3	4	...
`X when B`	1	1	3	3	5	...
`pre(X) when B`	nil	nil	2	2	4	...
`pre(X when B)`	nil	nil	1	1	3	...
`res1`	nil	nil	2	2	4	...
`res2`	nil	nil	1	1	3	...

Table 12: Example of Flows in Listing 16

As mentioned before, the idea was to evaluate the variables according to a specific clock making use of its state and history. So in this example, the equations defining `res1` and `res2` would be evaluated at each cycle as they are both sampled on the *basic clock*. Regardless of whether evaluating `res1` and `res2` according to clock `B` or the *basic clock*, one can see that the flows of `res1` and `res2` differs as shown in Table 12.
So it matters whether `pre` was applied first and then the timing operator `when` or the other way around. By following the approach described before, this information would have gone lost. This is due to the reason that expressions are not stored in the program's state.
To tackle this problem, all temporal expressions are rewritten such that the operands of `pre`, `current`, and `when` expressions are always stored in variables. For each operand of a `pre`, `current`, and `when` expression, it is stored in a temporary variable if it is not a variable. This means that the AST is changed by introducing temporary variables and equations defining them. Furthermore, the temporal expressions are rewritten such that they use the introduced variables. As explained later in Section 5.4.1, the operand of the `pre` operator which is now represented by a variable is always stored in the program's state. Here, it is obvious that their values in the previous cycle are also relevant for the calculation of the next cycle. Despite, all variables that are sampled on a clock slower than the *basic clock* are also stored in the program's state. They also include introduced variables for the operand of each `current` expression and the left-hand side operand of each `when` expression. Besides, each introduced equation is added to the node body at a position where all dependent variables in the current cycle are defined in a previous equation. This ensures that sequential interpretation of the node body is possible. As LUSTRE follows the *substitution principle*, it is ensured that there are no cyclic dependencies added by this approach. Furthermore, temporary variables are named starting with

`tmp_` followed by the counter for introduced temporary variables. If there are identifier clashes with existing identifier names, then the counter is incremented until there are no clashes.
Let `Expr1` be a temporal expression which is either equal to `pre(Expr2)` or `current(Expr2)` or `Expr2 when clock`. Again, let `Expr2` be an expression other than an identifier expression returning `n` values. Then `n` temporary variables and a single equation defining them are introduced. For introduced temporary variables `Tmp_1`, ..., `Tmp_n`, the equation `Tmp_1, ..., Tmp_n = Expr2` is added to the AST. Furthermore, `Expr1` is rewritten to `pre(Tmp_1, ..., Tmp_n)`, `current(Tmp_1, ..., Tmp_n)`, or `(Tmp_1, ..., Tmp_n) when clock` respectively. Afterwards, the resulting expression lists and the equation where `Expr1` is replaced are rewritten as explained before. This rewriting rule is applied to the expression `Expr2` recursively if it contains temporal expressions. Furthermore, a description is generated for each introduced variable describing the LUSTRE construct it originally replaces. This is added to the variables' descriptions which are also passed to both backends.

It is not necessary to store `->` expressions or its operands in temporary variables for accessing previous values, or values that are sampled on other clocks. `If-Then-Else` expressions and `->` expressions might also be rewritten due to another reason which is explained later in this section. Storing operands of `current`, `pre`, and `when` expressions in temporary variables also simplifies evaluation of nested `->` expressions such as `0 -> pre(1 -> pre(2 -> pre(x)))`. Compared to initial versions of this work, it is not necessary to store each clock's time in the state anymore. Furthermore, one does not need to calculate the upper bound of the time for each clock anymore to avoid an infinite state space. This is due to the reason that the nesting depth of `pre` operators is now a maximum of one. Rewriting temporal expressions also provides the opportunity to get rid of the clock's state and the clock's history containing values of variables completely. It could be reduced such that each clock stores the information whether the current cycle is the initialization step only. Furthermore, the history could be eliminated from the LUSTRE state completely. Instead, previous values are accessed using the previous state. According to LUSTRE2B, it is required to store the previous values in local variables within the *clock step operation*.

All in all, introducing temporary variables makes it possible to evaluate temporal expressions correctly. Even though temporary variables increase the number of variables, the size of the program's state and state space could be decreased. This even leads to better performance for simulation and model checking compared to initial versions of the interpreter and LUSTRE2B.

The rewritten node in Table 12 and its expression flows are as follows:

```
node n(B : bool) returns(res1 , res2 : int);
var X : int; tmp_0 : int; tmp_1, tmp_2, tmp_3 : int when B;
let
        tmp_3 =  pre(tmp_2);
        tmp_0 =  pre(X);
        X = 1 ->  pre(X) + 1;
        res2 =  current(tmp_3);
        tmp_1 = tmp_0 when B;
        tmp_2 = X when B;
        res1 =  current(tmp_1);
tel;
```

Listing 17: Example for Node Solving Problem of Evaluation of Temporal Expressions

Cycle	**1**	**2**	**3**	**4**	**5**	**...**
`X`	1	2	3	4	5	...
`B`	true	false	true	false	true	...
`tmp_0`	nil	1	2	3	4	...
`tmp_1`	nil	nil	2	2	4	..
`tmp_2`	1	1	3	3	5	...
`tmp_3`	nil	nil	1	1	3	...
`res1`	nil	nil	2	2	4	...
`res2`	nil	nil	1	1	3	...

Table 13: Example of Flows in Listing 17

Rewriting -> and If-Then-Else Expressions As mentioned before, `->` and `if-then-else` expressions might also be rewritten. The example in Listing 18 shows two nodes with the problem according to `->` and `if-then-else` expressions respectively. As one can see, the expression list cannot be rewritten due to the node call returning two values.

```
node node1(x,y : int) returns (a,b : int);
let
  (a,b) = (b, 1) -> node3(x, y);
tel;

node node2(x,y : int) returns (a,b : int);
let
  (a,b) = if x = y then (b, 1) else node3(x, y);
tel;


node node3(x,y : int) returns (a,b : int);
let ... tel;
```

Listing 18: Example for Node Describing Evaluation Problem of -> and If-Then-Else Expressions

The idea is to introduce temporary variables for operands of `->` and `if-then-else` expression as well.
Let `Expr` be a node call returning more than one value which is either an operand of the `->` operator, or the `then` expression or `else` expression of `if-then-else`. Again, let `n` be the number of returned values from `Expr`. Then `n` temporary variables and a single equation defining them are introduced. For introduced temporary variables `Tmp_1`, . . . ,

`Tmp_n`, the equation `Tmp_1,...,Tmp_n = Expr2` is added to the AST. Furthermore, `Expr` is rewritten to `(Tmp_1, ..., Tmp_n)`. Afterwards, the resulting expression lists and the equation where `Expr` is replaced are rewritten as explained before. This rewriting rule is applied to the parameters of the node call `Expr` recursively. Similar to introduced temporary variables for temporal expressions, there are also descriptions added for introduced variables for `->` and `if-then-else`. Rewritten nodes in Listing 18 are portrayed in Listing 19.
The same problem also exists for `pre`, `when`, and `current`. But as they are already rewritten due to another reason explained before, this problem is already solved.

```
node node1(x,y : int) returns (a,b : int);
var tmp_0, tmp_1 : int;
let
  (tmp_0, tmp_1) = node3(x, y);
  b = 1 -> tmp_1;
  a = b -> tmp_0;
tel;

node node2(x,y : int) returns (a,b : int);
var tmp_0, tmp_1 : int;
let
  (tmp_0, tmp_1) = node3(x, y);
  b = if x = y then 1 else tmp_1;
  a = if x = y then b else tmp_0;
tel;


node node3(x,y : int) returns (a,b : int);
let ... tel;
```

Listing 19: Rewritten -> and If-Then-Else Expressions in Listing 18

Checking Cycles in Equations After rewriting equations and expression lists, the resulting AST is checked for cyclic dependencies between variables. As mentioned in Section 2.4, the definition of equations follows the *substitution* and *definition principle*. Thus, the order of the equation does not affect the semantics of a node. There, variables can be defined depending on other variables in the current cycle.
A cyclic dependency between two variables exists if they are defined depending on each other. In this case, one could check that the equations defining both variables always have a fixed point, i.e. that there is exactly one solution for the system of equations. This is the undecidable *causality problem*. [11]
In contrast, this work follows the approach that a program is rejected at parsing if there are cyclic dependencies between variables. So, it is checked for each equation whether the variables on the right-hand side have already been defined. If this is fulfilled, then the equation already meets the requirements of LUSTRE. Otherwise, it must be checked whether the following equations define the variables on the right-hand side without depending on the left-hand side variables.
Similar to [11], this parser rejects the following equation system which has a cyclic dependency between `X` and `Y` even though there is a fixed point.

```
X = if C then Y else Z; Y = if C then Z else X;
```

The algorithm for checking cyclic dependencies tries to rearrange the node body such that: Each equation must hold the property that accessed variables on the right-hand side for the current cycle have already been defined in previous equations. This property must also be held by variables in assertions. Otherwise, the variables at the current cycle within assertions cannot be accessed at interpretation and evaluation also. "In the current cycle" means that these variables are not within `pre` expressions.
If there exists such an order, then there are no cyclic dependencies in the given node body. Otherwise, the node body contains cyclic dependencies of variables and thus the program is rejected. The resulting order of equations and assertions also makes it possible to interpret them sequentially. Instead of using the resulting order, it would also be possible to apply the *substitution principle* on the variables in the node body for parallel interpretation (see Section 5.4.2). So, this avoids the problem of interpreting equations and assertions that depend on variables that are not defined yet. Thus, the reordered node body is already a kind of AST optimization for both backends. So this step solves the following two tasks at the same time:

1. Check cyclic dependencies between variables
2. Change the order of equations and assertions such that the node body can be interpreted sequentially

When reordering the node body, each equation is checked whether the accessed variables on the right-hand side for the current cycle are defined in a previous equation yet. This is checked for variables within assertions, too. If these variables are already explored, then the equation or assertion is added to the resulting list for the node body and the counter is set to 0. Otherwise, it is added to the end of the list that must still be processed. In this case, the counter for checking cyclic dependencies is increased by one. If the counter is greater than the length of the node body part that must still be processed, then a cyclic dependency is detected and semantic checking fails here. The algorithm is shown below. The function `direct_depends(Expr)` denotes the variables `Expr` depends on directly. These are variables which are not within a `pre` operator in the given expression. Furthermore, `all(Pred, Lhs, Rhs)` is a function checking whether all elements in `Lhs` fulfils `Pred` according to `Rhs`. Note that the portrayed algorithm is imperative, while the implementation is declarative. It is only intended to describe how the idea of the algorithm works. Furthermore, note that all functions are applied without side effects i.e. without modifying the value of its operands.

Algorithm 2: Cycle Checking and Reordering Node Body

```
Input: Node Parameters $Params$, Node Body $Body$: $Ea_1$; ...; $Ea_N$;
Output: Resulting NodeBody $ResBody$: $Res\_Ea_1$; ...; $Res\_Ea_N$;

Bd := Body;
Counter := 0;
ResBody := [];
DefVariables := Params;
while Bd is not empty do
    Next_Ea := head(Bd);
    DependVariables := direct_depends(rhs(Next_Ea));
    if all(member, DependVariables, DefVariables) then
        ResBody := append(ResBody, [Next_Ea]);
        Counter := 0;
        DefVariables := append(DefVariables, lhs(Next_Ea));
        Bd := tail(Bd);
    else
        Counter := Counter + 1;
        Bd := append(tail(Bd), [Next_Ea]);
        if Counter > length(Bd) then
            throw error('Cycle detected');
        else
            _

return ResBody;
```

The resulting order of equations and assertions corresponds to the order if *topological sort* [27] was applied. Instead of implementing using a counter, it would also be possible to generate a dependency graph of variables similar to Section 5.3.3. Compared to the approach followed in this work, it would not be possible to reorder the node body and detect cyclic dependencies between variables in a single step.
As cycle checking of equations is part of semantic checking, it is done before interpretation. It would also be possible to detect cyclic dependencies at runtime. In this case, cyclic dependencies would lead to a deadlock which could be detected at model checking. However, this would lead to a lack of performance at runtime and better performance at parsing. As parsing is only applied once, both tasks are done in the parser.

Listing 20 and Listing 21 show different LUSTRE nodes having a direct dependency of the variable `a` on the variable `b`. Again, there are direct dependencies of `b` on `c` and `c` on `a`. So there is a cyclic dependency between all three variables.

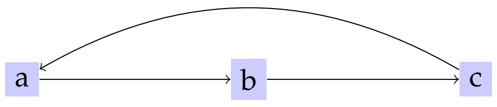
$$\begin{tikzcd} a \arrow[r] & b \arrow[r] & c \arrow[ll, bend right] \end{tikzcd}$$

Figure 10: Cycle Dependencies between Variables in Listing 20 and Listing 21

While the equation system in Listing 20 could be solved mathematically, the equation system in Listing 21 is not solvable. The solutions of the first example for `a`, `b`, and `c` are any integer value in which all three values are equal. But as this behavior is non-deterministic, it is semantically still wrong in LUSTRE. So semantic checking fails for both nodes.

```
node n(x : int) returns (a,b,c : int);
let
    a = b;
    b = c;
    c = a;
tel;
```

Listing 20: LUSTRE Node with Cyclic Definitions of Solvable Equation System

```
node n(x : int) returns (a,b,c : int);
let
    a = b + 1;
    b = c + 2;
    c = a;
tel;
```

Listing 21: LUSTRE Node with Cyclic Definitions of Unsolvable Equation System

In contrast, the example in Listing 22 shows a node where the order of equations must be rearranged before interpretation. There are direct dependencies of the variable `a` on `b`, `b` on `c`, and `c` on `x`. The latter variable is an input parameter.

$$\begin{tikzcd} a \arrow[r] & b \arrow[r] & c \arrow[r] & x \end{tikzcd}$$

Figure 11: Direct Dependencies between Variables in Listing 22

```
node n(x : int) returns (a,b,c : int);
let
    a = b + 1;
    b = c + 2;
    c = x + 4;
tel;
```

Listing 22: LUSTRE Program where Order of Equations Must Be Changed

For the example in Listing 22, the resulting order shown below enables sequential interpretation and sequential code generation.

```
node n(x : int) returns (a,b,c : int);
let
    c = x + 4;
    b = c + 2;
    a = b + 1;
tel;
```

Listing 23: Rewritten LUSTRE Program with Changed Order of Equations

5.4 Optimizing and Preprocessing

After applying lexing, parsing, and semantic checking successfully, some data are preprocessed from the resulting AST. This step determines which variables should represent the program's state. Furthermore, the list of clocks, and the clock hierarchy are generated during this step. The list of clocks is also used to represent the program's state. In contrast, the clock hierarchy is not part of the program's state, but it is required for interpretation and translation to B. Finally, the resulting AST and the preprocessed data are

then passed to both backends.
This section also evaluates other possibilities for optimizing the AST that are not followed in this work. Note that the result of semantic checking already generates an AST which can be interpreted sequentially, and where temporal expressions are rewritten such that they can be evaluated easily. So, some optimizations are already performed during semantic checking.

5.4.1 Preprocessing for State Representation

The optimized AST is used to preprocess information about the state representation for both backends. There, variables are divided into *state variables* and *non-state variables*. Furthermore, clocks, and the clock hierarchy are extracted from the given AST. This is done for each node in the given LUSTRE AST.
State variables correspond to variables in B that are defined in the `VARIABLES` clause. Thus, they represent the program's state. Again, *non-state variables* are similar to local variables in B, i.e. those that are defined in the `VAR` substitution. They are not relevant for the calculation in the following cycles and therefore do not represent the program's state. This is also the way how these variables are translated to B as explained later in Section 7. For the LUSTRE AST interpreter, this means that the *state variables* represent the program's state in contrast to the *non-state variables*.
LUSTRE variables are either input parameters, output parameters, or local variables. The *state variables* contain output parameters, clocks, variables that are sampled on a clock other than the *basic clock*, and variables that are used within a `pre` operator. Each verification node's output variable represents the safety properties that have to be verified. As they are often composed of several properties, variables on the right-hand side of the equation defining the output variable are also stored in the program's state. Variables representing the flow of a `pre`'s operand are always accessed for the calculation of the next cycle as result of Section 5.3.5. Again, flows that are sampled on a slower clock than the *basic clock* retain its value for the next cycle in the case that its clock is then inactive. Thus, they are always stored in a variable. This variable is then part of the program's state.
Each state stores the initialization status of each clock to evaluate safety properties, and `->` expressions. Therefore, preprocessing also determines which variables are clocks. The initialization status of a clock describes whether the next cycle is the clock's first cycle. In contrast, the clock hierarchy is used to determine whether a clock is active. Consider a clock `c2` that is sampled on `c1` with `c1` being sampled on the *basic clock*. Both clocks are passed as parameters to a LUSTRE node. If `c1` is set to `false`, then `c2` is also inactive even if `c2` is set to `true`. As the clock hierarchy does not change during the execution of the program, it is not part of the program's state.
So, the resulting preprocessed data that are passed to both backends are four AVL trees. Each AVL tree contains key-value pairs where a node name is mapped to its respective list of state variables, list of non-state variables, list of clocks, and clock hierarchy. The clock hierarchy is again a list of pairs representing a dependency graph between clocks. The *basic clock* is not part of the clock hierarchy as it is known to be the super clock of any other clock.

The following example shows a node with the input parameters `clock1`, `clock2`, `incr`, the output parameter n, and the local variable start. The temporal expressions are rewritten by adding the temporary variables tmp_0 and tmp_1 as shown in Listing 25.

```
node timed_incr(clock1 : bool; clock2 : bool when clock1; incr : int when clock2)
                returns (n : int when clock2);
var start : int when clock2;
let
  start = (0 when clock1) when clock2;
  n = (start -> pre(n)) + (start -> pre(incr));
tel;
```

Listing 24: Example for Preprocessing for State Representation

```
node timed_incr(clock1 : bool; clock2 : bool when clock1; incr : int when clock2)
                returns(n : int when clock2);
var start : int when clock2; tmp_0 : int; tmp_1 : int when clock1;
let
  tmp_0 = 0;
  tmp_1 = tmp_0 when clock1;
  start = tmp_1 when clock2;
  n = (start -> pre(n)) + (start -> pre(incr));
tel;
```

Listing 25: Optimized Example for Preprocessing for State Representation

Thus, the list of state variables is [clock1, tmp_1, clock2, n, start, incr], while the list of non-state variables is [tmp_0]. The resulting list of clocks is [$basic, clock1, clock2]. Again, the clock hierarchy for this node is [clock1-clock2].

5.4.2 Sequential Interpretation vs. Parallel Interpretation

As explained in Section 5.3.5, the AST is checked for cyclic dependencies between variables. The algorithm in this step tries to find an order of equations and assertions such that the program can be interpreted sequentially. If there exists such an order, then there are also no cyclic dependencies between variables in a cycle.
Instead of using the resulting order and interpreting the LUSTRE code sequentially, it would also be possible to apply the *substitution principle* on the AST to interpret in parallel. Variables in expressions would then be substituted such that expressions only contain parameter variables, or variables that are used within pre. This possibility has been rejected in this work as this would increase the number of operations, lead to more complex code, and also redundant code.

Consider the example portrayed in Listing 26. The equations in this node are ordered such that sequential interpretation is possible. It applies four arithmetic operations and a total number of eight lookups. Furthermore, four variables have to be updated during the interpretation of the node. Sequential interpretation means that the updated values might be accessed in the following equations and assertions for the calculation of the next cycle. E.g. the equation a = x stores the value of x in the variable a. The updated value of variable a is then used in the equation b = a + x.

```
node add(x,y,z : int) returns(a,b,c,d : int);
let
  a = x;
  b = a + x + pre(a);
  c = b + y;
  d = c + z;
tel;
```

Listing 26: Lustre Code where Sequential Interpretation is Possible

If one applies the *substitution principle* on this example, it would result in the AST with nine arithmetic operations, 13 lookups of variables, and four updated values. As shown in Listing 27, the resulting code is more complex and consists of a higher number of operations. This would also affect the performance of interpretation of the AST, and execution of the translated B models. According to model checking of B models, a translation with parallel substitutions might improve the performance. This could be analyzed in the future.

```
node add(x,y,z : int) returns(a,b,c,d : int);
let
  a = x;
  b = x + x + pre(a);
  c = x + x + pre(a) + y;
  d = x + x + pre(a) + y + z;
tel;
```

Listing 27: Applying *Substitution Principle* for Parallel Interpretation of Listing 26

It would also be required to expand node calls. On the one hand, this is necessary to apply B2PROGRAM on the translated B models as it does not support operation calls in parallel substitutions. On the other hand, this is also needed to access the return values of invoked nodes. As later in this section, the decision has been made not expand node calls.

5.4.3 Node Call Expansion

It would also be possible to expand node calls when optimizing the AST. Expanding means that the body of each invoked node is inlined into the invoking node. In this case, it would be necessary to handle collisions of identifier names across several nodes and even several node instances. E.g. it is possible to invoke a node more than once. Because each node call simulates an individual node, the variables of the node instances of the same node must then have different names. The inlined variable names can again have the same name as a variable in the invoking node. Expanding node calls into the invoking node's AST would result in a single node that is considered as a large component during simulation. It is then no longer easy to see which components make up the program and how they interact with each other.
According to this work, the decision has been made not to expand node calls. On the one hand, this diminishes the problem of handling collision of identifier names, especially in the translation to B. On the other hand, each node can then be viewed as an encapsulated component in the PROB animator for both interpretation using the LUSTRE AST interpreter and translation to B. The way how a state is portrayed in PROB is then easier

to reason about. To achieve this, the LUSTRE AST interpreter is implemented simulating each node call instances by invoking the predicate simulating a node recursively. Again, the translation to B aims to generate a machine file for each node. Machine instances for node calls are then included via the `INCLUDES` clause.
As a disadvantage, this work does not support variables on the left-hand side of an equation to be used within a node call on the corresponding right-hand side. Below, an example is shown where the equation `(S,T) = DUMMY(T,1)` contains a cyclic dependency between `S` and `T` which can be solved by node call expansion. [14]

```
node DUMMY(x,y : int) returns(a,b : int);
let
  a = x;
  b = y;
tel;
```

Listing 28: Example Where Node Call Expansion Is Required [14]

Concerning the future, one could apply node expansion in the case that it is necessary to solve cyclic dependencies between variables. As explained before, this leads to the disadvantage that an invoked node cannot be seen as an individual instance in the PROB animator anymore.

5.5 Error Handling

During the different parsing steps, many checkings are applied to the AST. This subsection describes how the different errors are handled. As the LUSTRE AST interpreter is integrated into PROB, all errors are shown using PROB's *error manager*. In contrast, LUSTRE2B is implemented as a console application. There, errors are printed out at the console.
Lexing fails at a specific position where the following character does not match the regular expression of any token. The corresponding error message contains the position and character.
The DCG Parser in Prolog is an LL(1)-parser that processes the token stream from left to right. During the parsing process, the parser stores the last token that was processed and the associated position. It also stores which token could have followed the last processed token. An error is then thrown with the position where parsing stops and a list of tokens that should have followed next.
When scope checking fails, errors are thrown with each of them containing the position. These error messages also contain the identifier of the invoked node or variable that is concerned. Despite scope errors, all other semantic errors are collected and thrown in the end. This makes it possible to provide better feedback about semantic errors. According to scope checking, this is not possible as both type checking and clock checking make use of the generated scope tables. Type checking, clock checking, and definition checking errors are thrown with each of them containing the position as well. For definition checking, the error message also contains the variables that are concerned. In contrast, errors in recursion checking do not contain position information. Instead, the error message contains the program's node dependency graph. For cycle checking, the resulting error message shows the part of the node body where the cyclic dependency is detected.

6 Interpretation in Prolog

After passing all parsing steps successfully, the optimized AST with semantic information is generated from the given LUSTRE program. Together with the preprocessed data and descriptions, it is passed to the LUSTRE AST interpreter for PROB. The LUSTRE AST interpreter is implemented containing the predicates `lustre_start/2`, `lustre_trans/3`, and `lustre_prop/2`. Loading a LUSTRE program requires implementing an XTL specification containing the predicates `start/1`, `trans/3`, and `prop/2` which dispatch on the other predicates named before. `lustre_start/2` expects the verification node's name as the first parameter, while the second parameter is passed to `start/1`. If the main node is not a verification node, then the first parameter should be set to `$none`.
This section describes the implementation of the interpreter in detail. The main tasks of the interpreter are interpretation and evaluation. Interpretation is applied to nodes, equations, assertions, and node calls. In contrast, the evaluation is applied to expressions which also include safety properties. Checking other properties such as nil checking of output variables, clocks, and assertion expressions are also tasks for the interpreter.
The complete semantics of the interpreter including auxiliary definitions and notations are portrayed in Appendix C. During this section, only the most important inference rules are shown. Each node's AST (via $ast(node)$) and the preprocessed data for each node ($svar_{node}$, $non-svar_{node}$, $clocks_{node}$, $chierarchy_{node}$) are globally accesible. Furthermore, each state during the interpretation is represented by the symbol σ. The respective *root state* and *abort state* are always denoted as σ_{root} and σ_{abort} explicitly. The *cut-off state* $\sigma_{cut-off}$ is only used within the calculation of the succeeding state. Again, σ_{state} denotes a state that is shown in the PROB animator i.e. a regular state after applying the initialization or a complete clock step. $\rightsquigarrow_{S;}$ denotes modification of a state after interpreting $S;$ (used in the form $\sigma \rightsquigarrow_{S;} \sigma'$). $\rightarrow_{\alpha(p_1,\ldots,p_n)}$ denotes modification of a state after applying an action α with given parameters (e.g. updating values or simulating node calls). $\Rightarrow$ denotes an evaluation without modifying the state (used in the form $\langle x, \sigma\rangle \Rightarrow E$, where x is a variable, σ a state and E the result). $\Rightarrow_\kappa$ denotes an evaluation in a specific context κ (e.g. checking whether or clock is active or checking safety properties). $\sigma \lhd\!\!- \{v \mapsto E\}$ overrides value of v to E in σ similar to the `override` operator in B.

6.1 Representation of A State

The initial state of each program is `Lustre Root` which is the state before choosing a node for simulation. This is realized with the implementation of the predicate `lustre_start/2`. After a node is chosen, the current state contains the name of the chosen node, the values of the state variables, the clocks' initialization statuses, and the invoked nodes' state. Thus, the corresponding state representation is `lustre_state(NodeName, StateVariables, Clocks, InvokedNodes)`.
The properties of the LUSTRE states are implemented using the predicate `lustre_prop/2`, which is also provided by the PROB interface. With this predicate, it is possible to display the program's state in PROB. The displayed state also contains the description of invoked nodes and introduced temporary variables.

`StateVariables` is an AVL tree that is initialized using the list of *state variables*. It stores key-value pairs where each *state variable* is mapped to its value. Initially, each variable is initialized to the value `nil`.
As explained in Section 5.3.5, rewriting temporal expressions makes it possible to get rid of the history completely. By introducing temporary variables, expressions are rewritten such that the nesting depth of `pre` operators is at a maximum of one. Thus, it is not necessary to store previous values of variables.
The *state variables'* values in the previous cycle are passed to the predicates for interpretation and evaluation at each cycle as well. This is required for the evaluation of `pre` expressions.
Before interpretation, the environment of *non-state variables* where each of them is mapped to `nil` is initialized for each node using the list of *non-state variables*. As explained in Section 5.4.1, the values of *non-state variables* are not required for the calculation of the following cycles and are thus not part of the program's state. So, the environment where each *non-state variable* is mapped to `nil` is referenced before the interpretation of each cycle.
`Clocks` is an AVL tree where each clock is mapped to its initialization status. It is generated from the list of clocks that is part of the preprocessed data. A clock's initialization status describes whether the clock's next cycle is the initialization cycle. As a result, each clock is initialized with the value `true`. The clocks' initialization statuses are required for the evaluation of `->` expressions. Furthermore, the clock hierarchy is also passed to the interpreter. It is used for interpretation but is not part of the program's state as it does not change during the simulation of the program.
`InvokedNodes` is an AVL tree storing key-value pairs where each node instance id is mapped to the corresponding node's state. As each invoked node is a subprogram, its state representation is also `lustre_state(NodeName, StateVariables, Clocks, InvokedNodes).`

Similar to scope tables, the environment is also represented using AVL trees. This is due to the reason that the operations on AVL trees lead to a better performance than lists, especially for a large number of entries. [24]

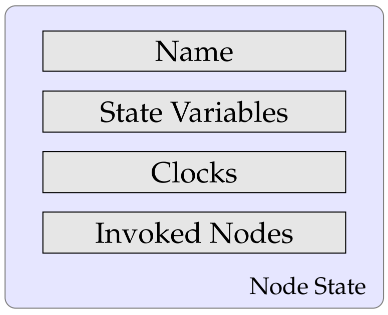

Figure 12: Representation of a LUSTRE Node State

6.2 Interpretation

As described before, the first transition is always the choice of a node for simulation. The operation performing this transition is of the form `choose_node(<Node>)`.

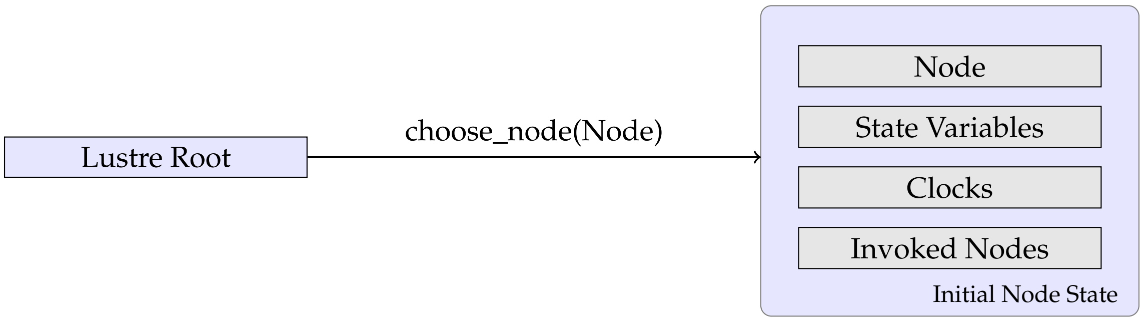

Figure 13: Initial Transition of a LUSTRE Program

Choosing a node initializes the node's environment where all *state variables* are assigned to `nil`. Besides, the initialization status of each clock is assigned to `true`, while each node call instance is initialized recursively. So the resulting state is the one before applying the first clock step. Furthermore, the chosen node is set as the main node. Outgoing from the root state, the inference rule $\rightarrow_{choose\_node(node)}$ is the entry point for interpretation. As a result of Section 5.4.3, the decision has been made not to expand node calls i.e. not to inline node calls in the invoking AST node.

$$\frac{\sigma_{root} \rightarrow_{init(ast)} \sigma_{init}}{\sigma_{root} \rightarrow_{choose\_node(node)} \sigma_{init}} \; ast(node) = ast$$

$$\frac{\sigma_{empty} \rightarrow_{init\_nodes(s_1;\ldots,s_n;)} \sigma_{init-nodes}}{\sigma_{root} \rightarrow_{init(\textbf{node}\ nodec\ldots;\ \textbf{let}\ s_1;\ldots,s_n;\ \textbf{tel};)} (nodec, svar_{nodec} \times \{nil\}, clocks_{nodec} \times \{true\}, \sigma_{init-nodes})}$$

After a node is selected, the following operations always perform a clock step with parameters for the chosen node. These operations are of the form `clock_step(Parameters)`. The parameters accept any values of their types satisfying the assertions. Performing a clock step changes the program's state. Recursively, it also changes each invoked node's state. Furthermore, each clock's initialization status is set to `false` once the clock has been active. The transition of an initialized state to a succeeding state is illustrated in Figure 14.

Interpretation of a LUSTRE node is done by interpreting the parameters, and the node body which contains equations, assertions, and node calls. Additionally, each clock's initialization status is updated. All this is implemented in the predicate `lustre_trans/3`. The corresponding entry point for interpretation is the inference rule $\rightarrow_{clock\_step([p_1=v_1,\ldots,p_n=v_n])}$.

Outgoing from a state in the PROB animator, it is extended by the environment of non-state variables, the environment of state variables in the current cycle, and a flag for cutting off the succeeding state which is initially set to false (rule for $\rightarrow_{prepare}$). As explained

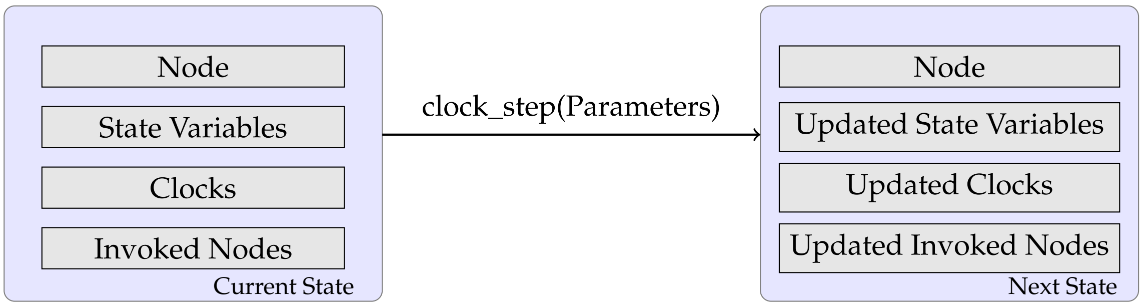

Figure 14: Clock Step Transition Changing Node State

before, the initial environment of non-state variables is accessed for each cycle where each *non-state variable* is mapped to `nil`. This is due to the reason that *non-state variables* do not represent the program's state. According to the calculation of the succeeding states, the environment of state variables in the current cycle is then the environment of the previous cycle. This is the way how `pre` expressions can be evaluated. The flag for cutting off the calculated state is set to true as soon as an assertion is evaluated to false during the interpretation.
If the interpretation of the parameters, or the node body results in the *abort state*, then it is also the succeeding state. The calculated state is then not cut off although the corresponding flag might be set to true. This is the reason why a flag for cutting off the calculated state is introduced. Again, if the interpretation results in a state other than the *abort state* with the flag for cutting off the calculated state being set to true, then an unreachable state is detected and is thus cut off.
In the case that there are no *aborts* during the interpretation and a reachable state is calculated, then this is also the succeeding state. Using the inference rule $\rightarrow_{state}$, the relevant environments for the state representation are taken and stored in the succeeding state. This means that the calculated environment of non-state variables, the environment of state variables in the previous cycle, and the flag for cutting off the calculated state are discarded.

$$\frac{\sigma_{state} \rightarrow_{sim(ast|p_1,\ldots,p_n|v_1,\ldots,v_n)} \sigma'_{state}}{\sigma_{state} \rightarrow_{clock\_step([p_1=v_1,\ldots,p_n=v_n])} \sigma'_{state}} \quad \begin{array}{c} node(\sigma_{state}) = nodec, ast(nodec) = ast, \\ parameters(ast) = p_1, \ldots, p_n, \\ \sigma'_{state} \neq \sigma_{cut-off} \end{array}$$

$$\frac{\begin{array}{c}\sigma_{state} \rightarrow_{prepare} \sigma', \sigma' \rightarrow_{sim-params(p_1,\ldots,p_n|v_1,\ldots,v_n)} \sigma'', \\ \sigma'' \rightarrow_{sim-body(s_1;\ldots;s_m;)} \sigma''', \sigma''' \rightarrow_{sim-clocks} \sigma'''', \sigma'''' \rightarrow_{state} \sigma'_{state}\end{array}}{\sigma_{state} \rightarrow_{sim(\textbf{node}\ nodec\ \ldots\ \textbf{let}\ s_1;\ldots;s_m;\ \textbf{tel};|p_1,\ldots,p_n|v_1,\ldots,v_n)} \sigma'_{state}} \quad ctf(\sigma''') = false$$

$$\frac{\sigma_{state} \rightarrow_{prepare} \sigma', \sigma' \rightarrow_{sim-params(p_1,\ldots,p_n|v_1,\ldots,v_n)} \sigma'', \sigma'' \rightarrow_{sim-body(s_1;\ldots;s_m;)} \sigma'''}{\sigma_{state} \rightarrow_{sim(\textbf{node}\ nodec\ \ldots\ \textbf{let}\ s_1;\ldots;s_m;\ \textbf{tel};|p_1,\ldots,p_n|v_1,\ldots,v_n)} \sigma_{cut-off}} \quad ctf(\sigma''') = true$$

$$\frac{\sigma_{state} \rightarrow_{prepare} \sigma', \sigma' \rightarrow_{sim-params(p_1,\ldots,p_n|v_1,\ldots,v_n)} \sigma'', \sigma'' \rightarrow_{sim-body(s_1;\ldots;s_m;)} \sigma_{abort}}{\sigma_{state} \rightarrow_{sim(\textbf{node}\ nodec\ \ldots\ \textbf{let}\ s_1;\ldots;s_m;\ \textbf{tel};|p_1,\ldots,p_n|v_1,\ldots,v_n)} \sigma_{abort}}$$

$$\frac{\sigma_{state} \rightarrow_{prepare} \sigma', \sigma' \rightarrow_{sim-params(p_1,\ldots,p_n|v_1,\ldots,v_n)} \sigma_{abort}}{\sigma_{state} \rightarrow_{sim(\textbf{node}\ nodec\ \ldots\ \textbf{let}\ s_1;\ldots;s_m;\ \textbf{tel};|p_1,\ldots,p_n|v_1,\ldots,v_n)} \sigma_{abort}}$$

The interpretation of the node body results in the *abort state* in one of the following cases: An assertion is evaluated to `nil` or a clock is defined as `nil`. Furthermore, invoked node instances might also be simulated to an *abort state* due to one of the two reasons mentioned before. This is part of the LUSTRE program's verification and is explained later in this section. Node calls might be applied with a clock being defined as `nil` if the corresponding variable is used as a clock in the invoked node only. This is the reason why the interpretation of parameters might lead to an *abort*.
For the interpretation of parameters, a list is expected where each parameter is assigned to a value. Each parameter is checked whether its clock is active. If its clock is active, then the parameter is updated to the passed value. Otherwise, it keeps its old value. This is realized with the inference rule for $\rightarrow_{update-param(v,e)}$ and $\rightarrow_{update(v,e)}$ as shown in Appendix C.3.4.

$$\frac{\sigma \rightarrow_{update-param(p_1,v_1)} \sigma', \ldots, \sigma^{(n-1)} \rightarrow_{update-param(p_n,v_n)} \sigma^{(n)}}{\sigma \rightarrow_{sim-params(p_1,\ldots,p_n|v_1,\ldots,v_n)} \sigma^{(n)}}$$

$$\frac{\ldots, \sigma^{(i)} \rightarrow_{update-param(p_{i+1},v_{i+1})} \sigma_{abort}}{\sigma \rightarrow_{sim-params(p_1,\ldots,p_n|v_1,\ldots,v_n)} \sigma_{abort}}$$

After interpreting the parameters with the given values, the node body is interpreted. There, the equations and assertions are interpreted sequentially. As described in Section 5.3.5, the order of the assertions and equations of each node is reordered such that sequential interpretation is possible.

$$\frac{\sigma \rightsquigarrow_{s_1;} \sigma', \ldots, \sigma^{(n-1)} \rightsquigarrow_{s_n;} \sigma^{(n)}}{\sigma \rightarrow_{sim-body(s_1;\ldots;s_n;)} \sigma^{(n)}}$$

$$\frac{\ldots \sigma^{(i)} \rightsquigarrow_{s_{i+1};} \sigma_{abort}}{\sigma \rightarrow_{sim-body(s_1;\ldots;s_n;)} \sigma_{abort}}$$

Finally, the initialization status of each clock is updated. For each clock, it is determined whether it is active in the current cycle. If this is the case, then its initialization status is set to `false`. Otherwise, it keeps its previous value.

$$\frac{\sigma \rightarrow_{sim-clock(c_1)} \sigma', \ldots, \sigma^{(n-1)} \rightsquigarrow_{sim-clock(c_n)} \sigma^{(n)}}{\sigma \rightarrow_{sim-clocks} \sigma^{(n)}} \ \mathrm{dom}(clocks(\sigma)) = \{c_1, \ldots, c_n\}$$

$$\frac{\langle c, \sigma \rangle \Rightarrow_{active} true}{\sigma \rightarrow_{sim-clock(c)} (\sigma \lhd\!\!+_{clock} \{c \mapsto false\})}$$

$$\frac{\langle c, \sigma \rangle \Rightarrow_{active} false}{\sigma \rightarrow_{sim-clock(c)} \sigma}$$

Figure 15 shows the initial state and a possible transition to a succeeding state outgoing from the node shown in Listing 3. Note that the lists are just used for illustration of the variables' values.

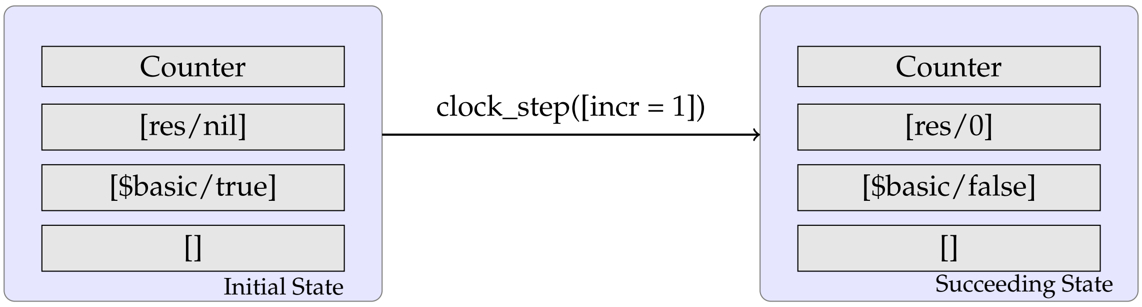

Figure 15: Example of Clock Step Transition Outgoing from Initial State and Listing 3

6.2.1 Equations

All node calls are within the expression on the right-hand side are interpreted first when interpreting an equation. This makes it possible to access the updated return values of the node calls. If there exists a node call on the right-hand side where its simulation results in an *abort state* then the interpretation of the equation results in an *abort state*. Note that the flag for cutting off the succeeding state is possibly set if there are assertion expressions in the node calls evaluated to `false`.
Let v_1, ..., v_n = E be an equation, let the simulation of the node calls in `E` result in a state other than the *abort state* and let e_1, ..., e_n be the evaluation of E. For each v_i it is then checked whether its clock is active. Similar to the interpretation of a parameter, it is also checked whether v_i is a clock and e_i is `nil`. In this case, the succeeding state is the *abort state*. Otherwise, the value of v_i is defined as e_i for the next cycle. If the clock of v_i is not active, then the variable keeps its old value. This is realized with the inference rule for $\rightarrow_{update(v,e)}$ (see Appendix C.3.4).

$$\frac{\sigma \rightarrow_{sim(E)} \sigma', \langle E, \sigma' \rangle \Rightarrow (e_1, \ldots, e_n), \sigma' \rightarrow_{update(v_1,e_1)} \sigma'', \ldots, \sigma^{(n)} \rightarrow_{update(v_n,e_n)} \sigma^{(n+1)}}{\sigma \rightsquigarrow_{v_1,\ldots,v_n=E;} \sigma^{(n+1)}}$$

$$\frac{\sigma \rightarrow_{sim(E)} \sigma', \langle E, \sigma' \rangle \Rightarrow (e_1, \ldots, e_n), \ldots, \sigma^{(i)} \rightarrow_{update(v_i,e_i)} \sigma_{abort}}{\sigma \rightsquigarrow_{v_1,\ldots,v_n=E;} \sigma_{abort}}$$

$$\frac{\sigma \rightarrow_{sim(E)} \sigma_{abort}}{\sigma \rightsquigarrow_{v_1,\ldots,v_n=E;} \sigma_{abort}}$$

So, equations are interpreted similarly to assignments in imperative programming languages with two side conditions: A variable's value is only modified if its clock is active. Furthermore, there is an *abort* if the variable is a clock that is defined to `nil` or if there is an *abort* in a node call. Node calls on the right-hand side of the equation can be seen as functions that modify the program's state.

6.2.2 Assertions

Node calls in assertions are interpreted first when interpreting an assertion. During the simulation of a LUSTRE program, all assertions are expected to be true.
Let `assert E` be an assertion and let `e` be the evaluation of `E`. As the assertion expression is sampled on the *basic clock*, it is interpreted each cycle. If `e` is `true`, then the assertion meets the assumption of the program and thus interpretation continues. In contrast, if `e` is `false`, then the flag for cutting off the calculated state is set to `true` if there are no *aborts*. Although the program's assumptions are not met, the interpretation of the node body proceeds until the end to detect *aborts*. If there is an *abort*, then this is also the succeeding state. Otherwise, the state is cut off. Cutting off the state means that the clock step operation cannot be applied with the given parameter values outgoing from the current state. Thus, the calculated state is not reachable. Again, if `e` is equal to `nil`, then the transition from the current state with the given parameter values results in an *abort state*. The succeeding state is also the *abort state* if the simulation of any node call in the assertion expression results in the *abort state*.
The interpretation of assertions only changes the node calls' states in the assertion expression or when the succeeding state is the *abort state*. Note that there are no differences between the interpretation of assertions in the main node and invoked nodes.

$$\frac{\sigma \rightarrow_{sim(E)} \sigma', \langle E, \sigma' \rangle \Rightarrow v}{\sigma \leadsto_{\textbf{assert}\ E;} (\sigma' \mathrel{\triangleleft\!\!+}_{ctf}\ c)}\ v \neq nil, c = (ctf(\sigma') = true\ \vee\ v = false)$$

$$\frac{\sigma \rightarrow_{sim(E)} \sigma', \langle E, \sigma' \rangle \Rightarrow nil}{\sigma \leadsto_{\textbf{assert}\ E;} \sigma_{abort}}$$

$$\frac{\sigma \rightarrow_{sim(E)} \sigma_{abort}}{\sigma \leadsto_{\textbf{assert}\ E;} \sigma_{abort}}$$

Listing 29 shows a node containing assertions which cut off states and transitions. This results in the state space shown in Figure 16. Note that `x` is a *non-state variable* which is not part of the program's state. It is just shown for illustration.

```
node AssertionExample(x : int) returns (r : int);
let
  assert x >= 0 and x <= 2;
  assert r >= 0 and r <= 2;
  r = x + 1;
tel;
```

Listing 29: Example for Interpretation of Assertions

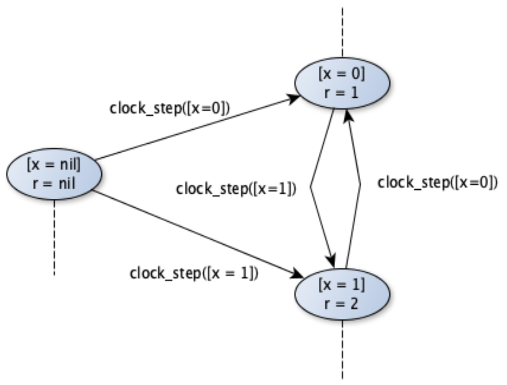

Figure 16: State Space where Assertions Cut Off States and Transitions

6.2.3 Node Calls

As explained before, each invoked node is an individual subprogram. When simulating an instance of an invoked node, node calls within its parameters are interpreted first. Similar to the fact that node calls in assertions and equations are interpreted first, this makes it possible to access the values of the invoked nodes within the parameters.
After interpreting the node calls in the parameters, the *basic clock* of the invoked node instance is determined. This might differ from the *basic clock* of the main node. As clocks are always declared before they are used, the invoked node instance's *basic clock* is the clock of its first parameter. If the clock of the invoked node instance is active then the simulation of the corresponding node instance is done like the simulation of the main node. Otherwise, the invoked node instance is not simulated in the current cycle.
The simulation of the given node call as well as the node calls in the parameters might lead to the *abort state*. As a result, the simulation of the whole program results in the *abort state*. Furthermore, if there are no *aborts* and at least one assertion within the node calls evaluated to `false`, then the resulting flag for cutting off the calculated state is set to `true`.

$$\frac{\begin{array}{c}\sigma \rightarrow_{sim(p_1,\ldots,p_n)} \sigma', \langle ck, \sigma' \rangle \Rightarrow_{active} true, \\ \langle (p_1,\ldots,p_n), \sigma' \rangle \Rightarrow (v_1,\ldots,v_n), \\ \sigma_{node} \rightarrow_{sim(ast|i_1,\ldots,i_n|v_1,\ldots,v_n)} \sigma'_{node}\end{array}}{\sigma \rightarrow_{sim(nodec(p_1,\ldots,p_n))} ((\sigma' \mathrel{\lhd\!\!-}_{ctf} c) \mathrel{\lhd\!\!-}_{node} \{NID \mapsto \sigma'_{node}\})} \quad \begin{array}{c} cl(p_1) = ck, ast(nodec) = ast, \\ parameters(ast) = (i_1,\ldots,i_n), \\ id(nodec(p_1,\ldots,p_n)) = NID, \\ NID \mapsto \sigma_{node} \in_{node} \sigma', \\ c = (ctf(\sigma') = true \lor \sigma'_{node} = \sigma_{cut-off}) \end{array}$$

$$\frac{\begin{array}{c}\sigma \rightarrow_{sim(p_1,\ldots,p_n)} \sigma', \langle ck, \sigma' \rangle \Rightarrow_{active} true, \\ \langle (p_1,\ldots,p_n), \sigma' \rangle \Rightarrow (v_1,\ldots,v_n), \sigma_{node} \rightarrow_{sim(ast|i_1,\ldots,i_n|v_1,\ldots,v_n)} \sigma_{abort}\end{array}}{\sigma \rightarrow_{sim(nodec(p_1,\ldots,p_n))} \sigma_{abort}} \quad \begin{array}{c} cl(p_1) = ck, ast(nodec) = ast, \\ parameters(ast) = (i_1,\ldots,i_n), \\ id(nodec(p_1,\ldots,p_n)) = NID, \\ NID \mapsto \sigma_{node} \in_{node} \sigma' \end{array}$$

$$\frac{\sigma \rightarrow_{sim(p_1,\ldots,p_n)} \sigma', \langle ck, \sigma' \rangle \Rightarrow_{active} false}{\sigma \rightarrow_{sim(node(p_1,\ldots,p_n))} \sigma'} \; cl(p_1) = ck$$

$$\frac{\sigma \rightarrow_{sim(p_1,\ldots,p_n)} \sigma_{abort}}{\sigma \rightarrow_{sim(node(p_1,\ldots,p_n))} \sigma_{abort}}$$

There are no differences between the state representation of the main node and an invoked node (see Figure 12).

6.3 Evaluation

LUSTRE expressions are either identifier expressions, literal expressions, data expressions, temporal expressions, or node calls. Safety properties are expressed as booleans in LUSTRE. The evaluation of expressions and safety properties is described in this subsection.

6.3.1 Identifier and Literal Expressions

Identifier and literal expressions are evaluated straightforward. While identifier expressions are evaluated by looking up the variable's value in the environment, literal expressions are evaluated to its literal value.

$$\frac{}{\langle x, \sigma \rangle \Rightarrow x} \; literal(x)$$

$$\frac{}{\langle x, \sigma \rangle \Rightarrow V} \; x \mapsto V \in_{var} \sigma$$

6.3.2 Data Expressions

Data expressions are either arithmetic, logical, comparison, or `if-then-else` expressions. The semantics for the evaluation of data expressions is portrayed in Appendix C.4.2.

Data expressions with its operands being not `nil` are evaluated straightforward. Most of them are evaluated imitating the usual behavior of these operators. Logical and `if-then-else` expressions are evaluated with short-circuiting. Furthermore, evaluation of `/` and `%` are implemented slightly more complicated to achieve LUSTRE's semantics. As division by zero is allowed as far as output variables, clocks, and assertions are not concerned, it is modeled as `nil` in this work.

For arithmetic and comparison expressions, the evaluation results in `nil` if any of the operands are evaluated to `nil`. This is also implemented with short-circuiting. The same behavior also applies to the evaluation of the logical expression with the operator `not`.

In contrast, `or`, `and`, and `if-then-else` expressions are evaluated differently. For the evaluation of an `or` expression, the result is `true` if the evaluation of one of its operands is `true`. This is also the case if the other operand is evaluated to `nil`. Similarly, the evaluation of an `and` expression results in `false` if one of its operands is evaluated to `false` even though the other operand is evaluated to `nil`. In all other cases, the evaluation of `or` and `and` expressions results in `nil` if at least one of the operands are evaluated to `nil`.
The evaluation of an `if-then-else` expression results in `nil` if the condition is evaluated to `nil`. In contrast, the result is always the evaluation of the `then` expression in the case that the condition is evaluated to `true`. Again, the evaluation of an `if-then-else` expression is the evaluation of the `else` expression if the condition is evaluated to `false`.

6.3.3 Temporal Expressions

Temporal expressions are expressions with any of the operators `pre`, `->`, `when`, or `current`. The operator `pre` is used to access previous values of an expression, whereas the operator `->` is used to define initial values of an expression flow. Combining `pre` and `->`, it is possible to define the first values of an expression explicitly. Again, `when` and `current` are used to sample an expression on a slower and faster clock respectively.

When A `when` expression consists of a clock variable on the right-hand side and expression on the left-hand side. As a result of Section 5.3.5, the expression on the left-hand side is always stored in a variable. Furthermore, the node body is rewritten such that the equations and assertions can be interpreted sequentially. So, the value of the left-hand side variable was updated during the interpretation of a previous equation. If a `when` expression is used within a `pre`, `current`, or `when` expression, then it is rewritten into another equation. Thus, a `when` expression is only within a data expression, a node call or a `->` expression in the optimized AST.

In the case that it is used within the node call parameters, then its clocking behavior is already implemented in the node call's simulation. If the clock of the `when` expression is the node call's *basic clock*, then the node call is never simulated with the clock being inactive. Otherwise, the invoked node instance is simulated with the updated value of the `when` expression.
A `when` expression could also be sampled on a clock that is passed through the node call. So, this clock is not the *basic clock* of the invoked node. If this clock is not active, then the interpretation of the invoked node does not make use of the value of the `when` expression. In contrast, the invoked node uses the updated value of the `when` expression if its clock is active.
If a `when` expression is not used within a node call parameter, then its evaluation depends on the equation it is part of. There, it is either a top-level expression or it might be used in a data or `->` expression on the right-hand side of the equation. Furthermore, `when` expressions within assertions are always within a node call or `current` in the supported subset of LUSTRE (before rewriting the AST). This is because an assertion expression's clock must always be the *basic clock*. In the case that it is used within `current`, the `when` expression is rewritten into another equation either.

During the interpretation of an equation, it is determined whether the corresponding clock is active. If the evaluation of the `when` expression depends on the equation it is part of, then it is not evaluated if its clock is not active. Otherwise, it is evaluated with the updated value of the left-hand side operand.

All in all, a `when` expression is always evaluated with the correct value even though the clock on the right-hand side is not taken into account. Thus, the evaluation of a `when` expression is the result of the lookup of the left-hand side variable in the environment.

$$\frac{}{\langle E \textbf{ when } C, \sigma\rangle \Rightarrow V}\; E \mapsto V \in_{var} \sigma$$

Current The `current` operator samples an expression on a faster clock in the hierarchy. For an expression `current(E)`, the evaluation results in the evaluation of `E` the last time its clock was active.
The operand of each `current` expression is always stored in a variable (see Section 5.3.5). Because of this and the fact that the `current` operator only activates an expression for a faster clock, its evaluation is done by looking up the variable representing its operand in the environment. So if the operand of a `current` expression is active in the current cycle, then it was updated in a previous equation as a result of Section 5.3.5. Otherwise, its value does not change during the current cycle. For both cases, it is ensured that the `current` expression is evaluated to its operand's value the last time it was active. Note that the operand of the `current` operator is always stored in the program's state as it is sampled on a clock other than the *basic clock*.

$$\frac{}{\langle \textbf{current}(E), \sigma\rangle \Rightarrow V}\; E \mapsto V \in_{var} \sigma$$

Pre Like `current` and `when` expressions, the operands of `pre` expressions are also stored in a variable (see Section 5.3.5). This provides the opportunity to get rid of nested `pre` operators in the optimized AST.
A `pre` expression is evaluated by looking up the variable representing its operand in the previous state. Note that a variable within a `pre` operator is always stored in the program's state.

$$\frac{}{\langle \textbf{pre}(E), \sigma\rangle \Rightarrow V}\; E \mapsto V \in_{his} \sigma$$

-> ("Followed-By") The program's state stores a flag for each clock describing whether the next cycle is the clock's first cycle. For the evaluation of a `->` expression, its clock and the corresponding flag are determined. If the next cycle of the clock is the initialization step, then the left-hand side of the `->` expression is evaluated as a result. Otherwise, the evaluation of the `->` expression results in the evaluation of the right-hand side operand.

$$\frac{\langle E_1, \sigma \rangle \Rightarrow V}{\langle E_1 \texttt{ -> } E_2, \sigma \rangle \Rightarrow V} \; cl(E_1 \rightarrow E_2) = C, C \mapsto true \in_{cl} \sigma$$

$$\frac{\langle E_2, \sigma \rangle \Rightarrow V}{\langle E_1 \texttt{ -> } E_2, \sigma \rangle \Rightarrow V} \; cl(E_1 \rightarrow E_2) = C, C \mapsto false \in_{cl} \sigma$$

Nested `pre` expressions containing `->` expressions are rewritten such that the interpreter only makes use of the environment of the previous and current cycle.

6.3.4 Node Calls

As each node returns a certain number of values, it is also used as an expression. After simulating a node corresponding to a node call, its output values are updated. For the evaluation, these values are read from the invoked node instance's state directly. With the state representation, there is a field containing the states of all invoked node instances.

$$\frac{}{\langle node(p_1, \ldots, p_n), \sigma \rangle \Rightarrow (v_1, \ldots, v_m)} \; \begin{array}{c} output(node(p_1, \ldots, p_n)) = (o_1, \ldots, o_m), \\ id(node(p_1, \ldots, p_n)) = NID, \\ NID \mapsto \sigma_{node} \in_{node} \sigma, \\ o_1 \mapsto v_1 \in_{svar} \sigma_{node}, \ldots, \\ o_m \mapsto v_m \in_{svar} \sigma_{node} \end{array}$$

6.3.5 Clock Activeness

For the evaluation and interpretation of some LUSTRE constructs, it is necessary to check whether a clock is active. To achieve this, the node's clock hierarchy is used. It is a list containing entries of the form $c_1 - c_2$ which means that the clock c_2 (transitively) depends on the clock c_1. Furthermore, the *basic clock* is not considered in the clock hierarchy as it is obvious that the *basic clock* is always active within the node it belongs to and any other clock depends on it. For any other clock, it is only active if this clock and all clocks it depends on hold the value *true*.

$$\frac{}{\langle c, \sigma \rangle \Rightarrow_{active} true} \; basic(c)$$

$$\frac{}{\langle c, \sigma \rangle \Rightarrow_{active} true} \; \begin{array}{c} \neg basic(c), c \mapsto true \in_{var} \sigma, \\ (\forall p.(p \in chierarchy_{node(\sigma)} \, \wedge \, p = c' \mapsto c) \implies c' \mapsto true \in_{var} \sigma) \end{array}$$

$$\frac{}{\langle c, \sigma \rangle \Rightarrow_{active} false} \; \begin{array}{c} \neg basic(c), (c \mapsto false \in_{var} \sigma \, \vee \\ (\exists p.(p \in chierarchy_{node(\sigma)} \, \wedge \, p = c' \mapsto c) \implies c' \mapsto false \in_{var} \sigma)) \end{array}$$

6.3.6 Verification

There are two issues when verifying a LUSTRE program: Applying nil checking on output parameters, clocks, and assertions, and verifying safety properties. Clocks and assertions must never be `nil` during the program's execution. This must also be the behavior for output parameters of the main node. The main node is the program that is chosen for simulation. So, nil checking is not applied to output parameters of node calls. Again, safety properties must always be `true` during the program's execution. Note that safety properties are only recognized if the corresponding node is annotated as a verification node. Each verification node is also a main node.
The state's properties are implemented using the predicate `lustre_prop/2`. As these verifications are applied to the program's state, erroneous states are implemented holding the property `unsafe` which is a built-in keyword for XTL specifications in the PROB interface. *Unsafe* states can be found at model checking.
Violations during nil checking of clocks, and assertions lead to an *abort state* which is also *unsafe*. Both cases are modeled as the *abort state* because it is not defined how the system should behave for them. Consider an assertion being evaluated to `nil`. This means that it is undefined whether the state is reachable and should thus be cut off. Again, consider a clock being evaluated to `nil`. In this case, this means that it is undefined whether the clock is active and whether its time steps forward in the step. As a result, the behavior of the variables it is sampled on this clock is also undefined.

$$\frac{}{\langle \sigma_{abort} \rangle \Rightarrow_{safe} false}$$

In contrast, nil values of output parameters, and safety property violations do not have undefined behavior as a consequence. Invoked nodes returning `nil` values can even be simulated without any violations continuously. Thus, they are modeled like an invariant. A main node's state where at least one output parameter holds the value `nil` once the output parameter's clock has been active is also *unsafe*.
Additionally, a verification node's state is *unsafe* if its safety property does not hold the value `true` once the output's clock has been active. Note that the safety property of a verification node is always its only return parameter.

$$\frac{\langle \sigma_{state} \rangle \Rightarrow_{output-nil(ast)} p}{\langle \sigma_{state} \rangle \Rightarrow_{safe} \neg p} \quad \begin{array}{c} node(\sigma_{state}) = nodec, \\ ast(nodec) = ast, \neg verif\_node(nodec) \end{array}$$

$$\frac{\langle \sigma_{state} \rangle \Rightarrow_{output-nil(ast)} p_1, \langle \sigma_{state} \rangle \Rightarrow_{safe-prop(ast)} p_2}{\langle \sigma_{state} \rangle \Rightarrow_{safe} \neg p_1 \ \wedge \ p_2} \quad \begin{array}{c} node(\sigma_{state}) = nodec, \\ ast(nodec) = ast, verif\_node(nodec) \end{array}$$

Nil Checking As mentioned before, an issue of verifying a LUSTRE program is ensuring that output parameters, clock values, and assertions are never `nil`.
For all output variables, it must be checked that its value is not `nil` once its clock has been active. This is applied to the main node's state.

$$\frac{\langle \sigma_{state}\rangle \Rightarrow_{output-nil(p_1)} false, \ldots, \langle \sigma_{state}\rangle \Rightarrow_{output-nil(p_n)} false}{\langle \sigma_{state}\rangle \Rightarrow_{output-nil(\textbf{node} \ldots \textbf{returns} (o_1;\ldots;o_n) \ldots)} false} \; oparameters(o_1;\ldots;o_n) = p_1,\ldots,p_n$$

$$\frac{\ldots, \langle \sigma_{state}\rangle \Rightarrow_{output-nil(p_i)} true}{\langle \sigma_{state}\rangle \Rightarrow_{output-nil(\textbf{node} \ldots \textbf{returns} (o_1;\ldots;o_n) \ldots)} true} \; oparameters(o_1;\ldots;o_n) = p_1,\ldots,p_n$$

$$\frac{}{\langle \sigma_{state}\rangle \Rightarrow_{output-nil(o)} true} \; clock(o) = c, o \mapsto nil \in_{svar} \sigma_{state}, c \mapsto false \in_{clock} \sigma_{state}$$

$$\frac{}{\langle \sigma_{state}\rangle \Rightarrow_{output-nil(o)} false} \; clock(o) = c, ((o \mapsto V \in_{svar} \sigma_{state} \wedge V \neq nil) \;\vee\; (c \mapsto true \in_{clock} \sigma_{state}))$$

Again, for all clock variables, it must also be checked that it is not defined as `nil` under the condition that its clock is active. Within the semantics for updating values, the inference rule $\Rightarrow_{c-nil}$ is used. The rule expects a variable, the value it should be assigned to, and the program's state. If the variable is a clock that is intended to be defined as `nil`, then $\Rightarrow_{c-nil}$ is evaluated to `true`. This information is then used to transfer to the *abort state*.

$$\frac{}{\langle v, e, \sigma\rangle \Rightarrow_{c-nil} true} \; v \in \mathrm{dom}(clocks(\sigma)), e = nil$$

$$\frac{}{\langle v, e, \sigma\rangle \Rightarrow_{c-nil} false} \; v \in \mathrm{dom}(clocks(\sigma)), e \neq nil$$

$$\frac{}{\langle v, e, \sigma\rangle \Rightarrow_{c-nil} false} \; v \notin \mathrm{dom}(clocks(\sigma))$$

Furthermore, it must hold that each assertion is not evaluated to `nil` during the program's execution. If an assertion is evaluated to `nil`, then the succeeding state is also the *abort state*. This is implemented in the interpretation of an assertion.
Aborts can also occur in invoked node instances. In this case, the invoking node also raises an *abort* error. This is realized in the interpretation of node calls.

Safety Properties Safety properties in LUSTRE can be expressed as boolean expressions. They are used in *verification nodes* which are nodes containing a single output parameter representing the composition of safety properties. In the LUSTRE language itself, *verification nodes* are also programs. So it is not classified in the language which nodes are *verification nodes*. Each *verification node* must have only one return parameter storing a boolean value. But this does not mean that each node with one return parameter of the

type boolean is also a *verification node*. This work enforces the verification node's name to be passed to `lustre_start/2` as the first argument in a LUSTRE program's loading file.
It must then be checked that the safety properties are always `true` under the condition that the assertions are `true` once its clock has been active. The interpretation of the assertions ensures that states are cut off where an assertion is evaluated to `false`. So, a state is only reachable if it holds the program's *assumptions* which are encoded in the assertions.

$$\frac{}{\langle\sigma_{state}\rangle \Rightarrow_{safe-prop(\textbf{node} \,\ldots\, \textbf{returns}(o)\ldots)} true} \quad \begin{array}{c} oparameters(o) = p, \\ clock(p) = c, ((p \mapsto true \in_{svar} \sigma_{state}) \vee \\ (c \mapsto true \in_{clock} \sigma_{state})) \end{array}$$

$$\frac{}{\langle\sigma_{state}\rangle \Rightarrow_{safe-prop(\textbf{node} \,\ldots\, \textbf{returns}(o)\ldots)} false} \quad \begin{array}{c} oparameters(o) = p, \\ clock(p) = c, p \mapsto V \in_{svar} \sigma_{state}, \\ V \neq true, c \mapsto false \in_{clock} \sigma_{state} \end{array}$$

7 Translation to B

Another approach that is followed in this work is translating LUSTRE programs to B for simulation and verification in PROB. Furthermore, this work also aims code generation to Java and C++ using B2PROGRAM. To achieve this, it targets translation to a subset of B that is supported by B2PROGRAM.
Similar to the LUSTRE AST interpreter, LUSTRE2B expects the optimized semantic AST and the preprocessed data after passing all steps in the frontend. Furthermore, descriptions for introduced temporary variables and node calls are passed to LUSTRE2B as well. They are generated as comments in the generated B files.
As each LUSTRE node is an individual program that might be invoked by other nodes, it can be seen as an individual component in B. Thus, each LUSTRE node is translated to a B machine as shown in Figure 17. The machine's name is set to the node's name preceded by the prefix `M_`. As a result, the generated machine file always starts with this prefix. This is required to translate nodes whose names start with an *underscore* as identifiers cannot start with an *underscore* in B.
Instead of generating a machine file for each node, it would also be possible to expand each node call. Expanding means that the code of each invoked node is inlined into the invoking node. As a result of Section 5.4.3, the decision has been made not to follow this approach.

Each generated B machine *sees* the machine `LibraryLustre` via the `SEES` clause. `LibraryLustre` is developed for LUSTRE2B and contains the B implementation for the data operators and `->`. These operators are declared as constants and defined as *lambda expressions* in the `PROPERTIES` clause. Other temporal operators are translated differently. With this approach, the translation of expressions to B can be simplified which will be explained in this section.
Machines with node calls *include* instances of the corresponding machines via the

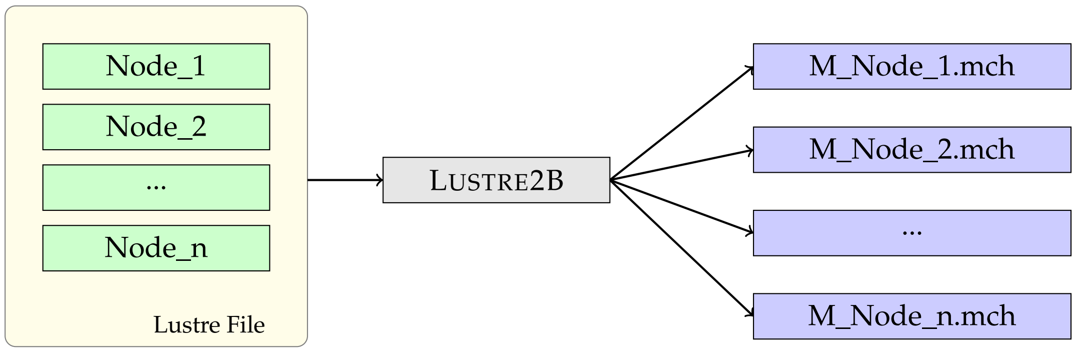

Figure 17: Translation of Lustre File with Many Nodes

`INCLUDES` clause. The inclusion and simulation of invoked nodes will be explained in this section as well.

First, it will be discussed how LUSTRE variables and values are represented in the translation to B. This leads to the implementation of the LUSTRE operators in `LibraryLustre`. Furthermore, this section describes how LUSTRE2B works. The denotational semantics of the translation to B is portrayed in Appendix D. Only the most important translation rules are shown in this section. A few examples of translating LUSTRE to B are portrayed in Appendix E. The translation rules always refer to the current node. Here, the preprocessed data of the corresponding node (list of *state variables*, list of *non-state variables*, list of clocks, and the clock hierarchy) are globally accessible. They are denoted as `svar`, `non-svar`, `clocks`, and `chierarchy` respectively. Furthermore, the current node's name is globally accessible as `node_name` for simplification of the translation rules. Again, a translation rule is denoted by $[\![]\!]$. As portrayed in Appendix D, there are also auxiliary definitions including some functions.

7.1 Representation of Lustre Variables and Values

There are several ways to represent LUSTRE variables and values in B. Each of them has advantages and disadvantages. The main difference between LUSTRE and B is that B does not support uninitialized i.e. `nil` values.

The examples that are portrayed in this subsection contains the variables `x`, `y`, and `z`. While `x` and `y` are of the type integer, `z` is of the type boolean. Furthermore, `x` holds the value `1`, while `y` holds `nil` as its value. Again, the variable `z` stores the value `TRUE`.

For now, the only supported types of LUSTRE2B are integer and boolean. LUSTRE also supports user-defined types and real numbers that both could be implemented as future work. For the translation of real numbers, it would require to extend the B language.
The intuitive option is translating LUSTRE values to the corresponding B values i.e. LUSTRE integer are translated to B integers and LUSTRE booleans are translated to B booleans. But following this approach, it is not possible to model `nil` values.

```
x ⇝ 1
y ⇝ ???
z ⇝ TRUE

1 ⇝ 1
true ⇝ TRUE
nil ⇝ ???
```

Listing 30: One-to-One Representation of Lustre Variables and Values in B

So, LUSTRE values have to be represented differently. Furthermore, most LUSTRE operators are implemented in a library `LibraryLustre` imitating the LUSTRE semantics. The implementation of `LibraryLustre` takes `nil` values for each implemented operator into account.
Another possibility is generating a variable for each type that appears in the LUSTRE node. For this approach, it would be necessary to declare a set containing an element for each variable. This variable then stores the values of all variables of the type it represents. So, it is a relation that maps elements from the variable set to its values. If a set element is not in the domain of this variable, then its value is `nil`.

```
current_state_int ⇝ {x ↦ 1}
current_state_bool ⇝ {z ↦ TRUE}

1 ⇝ 1
true ⇝ TRUE
nil ⇝ –
```

Listing 31: Representation of Lustre Variables and Values in B with Variable for Each Type

However, this approach has some disadvantages: If more types are supported for the translation to B in the future, then such a variable must be defined for each of these types. This becomes complicated as soon as enum, record, and array types are implemented. Variables in the PROB animator are not listed one after the other by following this approach. Instead, one has to look within the relation of the respective type for the value of the variable. This makes it hard to reason about the program especially when there are many variables of the same type. If one is looking for a variable holding `nil`, then the corresponding element is even not within the relation.
This also leads to another problem: Consider the additions `x + y` and `x + 1`. For the first addition, the invocation of the implemented LUSTRE operator would be of the form `l_plus(current_state_int, x, y)`. In contrast, the second addition would be of the form `l_plus(current_state_int, x, 1)`. As one can see, the operands are from different types. So, it would be necessary to implement a function for each combination of literals and variable set elements for each operator.
Another possibility is modeling `nil` values as the $\varnothing$ and other values as a singleton set storing the value. This would make it possible to use the `MU` operator in `CHOOSE.def` to reference the value. But this approach is also not suitable to aims code generation using B2PROGRAM as the `DEFINITIONS` clause is not supported.

```
x ⇝ {1}
y ⇝ ∅
z ⇝ {TRUE}

1 ⇝ {1}
true ⇝ {TRUE}
nil ⇝ ∅
```

Listing 32: Representation of Lustre Variables and Values in B with MU Operator

For the final solution, a similar idea is considered. Each LUSTRE value is represented as a relation that stores a maximum of one element. Relations in B are sets of tuples. Again, functions are relations with the well-definedness condition that each element in the domain is mapped to a maximum of one element in the range. `nil` values are then modeled as ∅, too. In contrast, all other values are represented as $\{ref \mapsto val\}$. The element `ref` is the only element in the singleton set `REF` that is defined in `LibraryLustre`. Thus, the translated integer and boolean expressions are of the type $\mathbb{P}$(REF $\times$ $\mathbb{Z}$) and $\mathbb{P}$(REF $\times$ BOOL) respectively. With this approach, it is possible to access the value via a function call on `ref`. In contrast to the `MU` operator, this is supported by B2PROGRAM.
As the operators =, <>, `if-then-else`, and `->` can be used for any type, there is the problem they must be implemented for each type. This is due to the reason that B does not support generic types for variables and constants in the `PROPERTIES` clause. Instead of using the `PROPERTIES` clause, it would also be possible to implement these operators in the `DEFINITIONS` clause. But as this work also aims code generation to a subset of B that makes the use of B2PROGRAM feasible, the operators are implemented in the `PROPERTIES` clause. For arrays, records, and user-defined types, it would be necessary to generate the implementation of the operators for these types. They could then be provided in another library that is *seen* by all generated B machines. In contrast to the second idea presented in this section, there are no problems according to the implementation of the operators.

```
x ⇝ {ref ↦ 1}
y ⇝ ∅
z ⇝ {ref ↦ TRUE}

1 ⇝ {ref ↦ 1}
true ⇝ {ref ↦ TRUE}
nil ⇝ ∅
```

Listing 33: Representation of Lustre Variables and Values with Reference Set

7.2 Implementation of Lustre Operators

Arithmetic, comparison, logical as well as `if-then-else` and `->` operators are implemented as B functions imitating the semantics in LUSTRE. As explained before, they are declared as constants and implemented as *lambda expressions* in the `PROPERTIES` clause of the machine `LibraryLustre`. `LibraryLustre` is *seen* by all generated B machines. This makes it possible to translate these operators one-by-one to the invocation of the corresponding B functions. Otherwise, it would be necessary to generate the implementation of these operators every time they are used. However, this would lead to duplicated and also more complicated code.

The temporal operators `pre`, `current`, and `when` are not implemented. They are translated differently as explained in Section 7.8.3. As explained before, `LibraryLustre` also contains a singleton set `REF` with the element `ref`.
Furthermore, the respective sets `LUSTRE_INT`, `LUSTRE_INT_NOT_NIL`, `LUSTRE_BOOL`, and `LUSTRE_BOOL_NOT_NIL` are defined for both types that are supported in this work. `LUSTRE_INT` is the set of integer values in the LUSTRE representation. Similarly, `LUSTRE_BOOL` is the set of boolean values in the representation in LUSTRE. Again, `LUSTRE_INT_NOT_NIL` and `LUSTRE_BOOL_NOT_NIL` are the sets of non-nil values for both types in the LUSTRE representation.

Type	**Definition**
LUSTRE_INT	$\{x \mid x \in \mathbb{P}(\mathrm{REF} \times \mathbb{Z})\ \&\ \mathrm{card}(x) \leq 1\}$
LUSTRE_BOOL	$\{x \mid x \in \mathbb{P}(\mathrm{REF} \times \mathrm{BOOL})\ \&\ \mathrm{card}(x) \leq 1\}$
LUSTRE_INT_NOT_NIL	$\{x \mid x \in \mathbb{P}(\mathrm{REF} \times \mathbb{Z})\ \&\ \mathrm{card}(x) = 1\}$
LUSTRE_BOOL_NOT_NIL	$\{x \mid x \in \mathbb{P}(\mathrm{REF} \times \mathrm{BOOL})\ \&\ \mathrm{card}(x) = 1\}$

Table 14: Implementation of LUSTRE Integer and Boolean Types

The implemented LUSTRE operators in B and the corresponding types are portrayed in Table 15. As `=`, `<>`, `if-then-else`, and `->` can be used for any type, they are implemented for both supported types in this work. Furthermore, data operators and `->` are implemented imitating the behavior described in Section 6.3.2 and Section 6.3.3 respectively. The functions' operands and output are the values with the LUSTRE representation explained before. Corresponding values are accessed via a function call on `ref` in B.

B	**Function**
l_plus l_minus l_multiply l_divide	LUSTRE_INT × LUSTRE_INT → LUSTRE_INT
l_unary_minus	LUSTRE_INT → LUSTRE_INT
l_less l_less_equal l_greater l_greater_equal l_equal_integer l_unequal_integer	LUSTRE_INT × LUSTRE_INT → LUSTRE_BOOL
l_equal_boolean l_unequal_boolean l_and l_or	LUSTRE_BOOL × LUSTRE_BOOL → LUSTRE_BOOL
l_not	LUSTRE_BOOL → LUSTRE_BOOL
l_ite_integer	LUSTRE_BOOL × LUSTRE_INT × LUSTRE_INT → LUSTRE_INT
l_ite_boolean	LUSTRE_BOOL × LUSTRE_BOOL × LUSTRE_BOOL → LUSTRE_BOOL
l_fby_integer	BOOL × LUSTRE_INT × LUSTRE_INT → LUSTRE_INT
l_fby_boolean	BOOL × LUSTRE_BOOL × LUSTRE_BOOL → LUSTRE_BOOL

Table 15: Lustre Operators in B

Although `->` is a binary operator, the corresponding implemented functions expect three operands. The first operand is the initialization status of the expression's clock. In contrast, the second and third operands are the respective left-hand side and right-hand side operand. As explained later in this section, the initialization status of each clock is modeled as a variable in the translation to B. It describes for a clock whether the next cycle is the clock's first cycle. For a function call `l_fby_integer(x,y,z)`, `y` is returned if `x` is `true`. Otherwise, `z` is returned. The similar behavior is implemented for `l_fby_boolean(x,y,z)`.

7.3 Translation of Node

Each node is translated to a B machine. A LUSTRE node is composed of its name, input parameters, output parameters, local variables, and a body containing equations and assertions.
When translating a node to a machine, the machine's name is set to the node's name preceded by the prefix `M_`. As a result, the generated machine file always starts with this prefix. This is required to translate nodes whose names start with an *underscore* because identifiers cannot start with an *underscore* in B.

$$machine\_name(node) = \ format(\text{“}M\_\%s\text{“}, node)$$

LUSTRE2B also generates the `SEES`, `VARIABLES`, `INVARIANT`, `INITIALISATION`, and `OPERATIONS` clause. If there are node calls in the given node body, then an `INCLUDES` clause is generated containing the inclusion of machines representing node call instances. As mentioned before, each generated B machine *sees* `LibraryLustre`.
`VARIABLES` is generated using the list of *state variables* and the list of clocks only. There, the optimized AST is not taken into account.
Again, these variables are typed in the `INVARIANT` clause. Despite typing predicates, it also contains predicates for nil checking of output parameters, and safety properties. For the generation of `INVARIANT`, it is necessary to access semantic information of the *state variables*. To achieve this, the declarations AST nodes are passed to the predicate generating `INVARIANT`.
All declared variables in `VARIABLES` are initialized in the `INITIALISATION` clause. Similar to `VARIABLES`, it is generated using the list of *state variables* and the list of clocks only.
To simulate a node, a *clock step operation* is generated in the `OPERATIONS` using the AST nodes for input parameters, and the node body.

$$[\![\textbf{node}\ nodec(input)\ \textbf{returns}(output);\ \textbf{var}\ locals;\ \textbf{let}\ body\ \textbf{tel};]\!]_{tr} =$$
$$\begin{cases} \textbf{MACHINE}\ machine - name \\ [\![body]\!]_{includes} \\ \textbf{SEES}\ LibraryLustre \\ \textbf{VARIABLES}\ VARS \\ \textbf{INVARIANT}\ [\![input, output, locals]\!]_{invariant} \\ \textbf{INITIALISATION}\ INIT \\ \textbf{OPERATIONS}\ [\![input, body]\!]_{clock-step} \\ \textbf{END} \end{cases} \text{if}\ \begin{array}{l} machine - name = machine\_name(nodec), \\ VARS = variables, \\ INIT = init \end{array}$$

7.4 Generation of INCLUDES Clause

As each node call simulates an individual node instance, it is necessary to generate an `INCLUDES` clause where all these instances are included. If there are no node calls in the node, then no `INCLUDES` clause is generated for the machine.

To achieve this, the whole node body is visited recursively with the task to determine all node calls. Each time a node call is visited, an instance with the given id and node name is generated. The complete translation rules for generating INCLUDES is portrayed in Appendix D.3.

$$[\![s_1; \dots; s_n;]\!]_{includes} = \epsilon \text{ if } [\![s_1;]\!]_{includes} = \epsilon, \dots, [\![s_n;]\!]_{includes} = \epsilon$$

$$\begin{gathered}[\![s_1; \dots; s_n;]\!]_{includes} = \textbf{INCLUDES } [\![s_1;]\!]_{includes} \circ, \dots \circ, [\![s_n;]\!]_{includes} \\ \text{if } [\![s_1;]\!]_{includes} \neq \epsilon \textit{ or } \dots \textit{ or } [\![s_n;]\!]_{includes} \neq \epsilon\end{gathered}$$

$$\begin{gathered}[\![node(p_1, \dots, p_n)]\!]_{includes} = [\![p_1, \dots, p_n]\!]_{includes} \circ, ref \\ \text{if } nid = id(node(p_1, \dots, p_n)),\ ref = reference\_name(nid, node)\end{gathered}$$

7.5 Generation of VARIABLES Clause

The VARIABLES clause is generated using the list of *state variables* and the list of clocks. VARIABLES consists of *state variables*, variables for each clock's initialization status, and a variable for assertions named assert_ok. As explained in Section 5.4.1, *state* variables are variables that represent the program's state. The initialization status of a clock describes whether the next cycle is the clock's first cycle. It might be used within safety properties, nil checking of output parameters, and -> expressions. Furthermore, they are updated at the end of a cycle. Again, assert_ok describes whether the program's assumptions are met. It is used to cut off unreachable states. The reason why such a variable is required is explained later in the translation of assertions. All these variables are typed in the INVARIANT and initialized in the INITIALISATION.

$$variables = \begin{cases} assert\_ok, \\ [\![c_1]\!]_{i-status}, \dots, [\![c_n]\!]_{i-status}, \\ [\![s_1]\!]_{var}, \dots, [\![s_n]\!]_{var} \end{cases} \text{if } \begin{array}{l} clocks = c_1, \dots, c_n, \\ svar = s_1, \dots, s_n \end{array}$$

$$[\![v]\!]_{var} = var \text{ if } var = var\_to\_b(v)$$

$$[\![c]\!]_{i-status} = var \text{ if } var = i - status(c)$$

Each *state variable* starts with the prefix var_, while each *initialization status variable* starts with *is_initialisation*. This is required to ensure that generated variables do not start with an *underscore* as identifiers are not allowed to start with an *underscore* in B.

$$var\_to\_b(var) = format(\text{``}var\_\%s\text{``}, var)$$

$$i - status(c) := \begin{cases} \text{``}is\_initialisation\text{``}, & c = \$basic \\ format(\text{``}is\_initialisation\_\%s\text{``}, c), & c \neq \$basic \end{cases}$$

7.6 Generation of INVARIANT Clause

The INVARIANT clause contains predicates typing the variables that are declared in the VARIABLES clause. There are also predicates for nil checking of output variables if the corresponding node is the main node. Output values of the main node are the output of the complete program. When a main node's output parameter yields nil, then it is still possible to simulate the program continuously. Invoked nodes are allowed to return nil values in contrast to the main node. In addition to typing, and nil checking predicates, also a predicate for checking the safety property is generated if this is a *verification node*. Note that each *verification node* is also a main node.

$$[\![input, output, locals]\!]_{invariant} = [\![input, output, locals]\!]_{typing} \text{ if } not(main\_node(node\_name))$$

$$[\![input, output, locals]\!]_{invariant} = \begin{cases} [\![input, output, locals]\!]_{typing} \ \& \\ [\![output]\!]_{not-nil} \end{cases} \text{ if } main\_node(node\_name), not(verif\_node(node\_name))$$

$$[\![input, output, locals]\!]_{invariant} = \begin{cases} [\![input, output, locals]\!]_{typing} \ \& \\ [\![output]\!]_{not-nil} \ \& \\ [\![output]\!]_{safety} \end{cases} \text{ if } verif\_node(node\_name)$$

Clocks and assertions must also be checked for nil values. nil values in clocks and assertions lead to undefined behavior of the program. According to assertions, it is not defined whether the calculated state should be cut off. Again, for clocks, it is not defined whether the clock is active and steps forward. This is the reason why they are not generated in the INVARIANT clause. Instead, nil values of clocks and assertions lead to an *abort state* which is realized with the PRE substitution in B. A PRE substitution is used checking that a clock holds a value other than $\varnothing$ after it is assigned to a value. This is modeled in the translation of an equation. Similarly, a PRE substitution is used when translating an assertion. It checks that the value of the translated assertion expression is not equal $\varnothing$. Unlike PRE, SELECT cannot be used to model an *abort*. Furthermore, ASSERT yields the same behavior as PRE according to simulation. But for verification, it seems that PRE provides more information to symbolic model checking than ASSERT.

7.6.1 Typing Predicates

Variables that are declared in the VARIABLES clause are typed in the INVARIANT. As explained before, VARIABLES only consists of the variables that represent the program's state. These are *state variables*, variables describing the clocks' initialization statuses, and a variable for assertions named assert_ok. The variable assert_ok is typed as BOOL.

$$[\![input, output, locals]\!]_{typing} = \begin{cases} assert\_ok \in \text{BOOL} \ \& \\ [\![c_1]\!]_{i-status} \in \text{BOOL} \\ \& \ldots \& \\ [\![c_n]\!]_{i-status} \in \text{BOOL} \ \& \\ [\![input]\!]_{typing} \circ_{\&} [\![output]\!]_{typing} \circ_{\&} \\ [\![locals]\!]_{typing} \end{cases} \text{ if } clocks = c_1, \ldots, c_n$$

During the generating of the typing predicates, the declarations AST nodes are visited to access type information of the *state variables*. For each declared variable, a corresponding typing predicate is generated if it is a *state variable*. Furthermore, the initialization status of each clock is typed as BOOL.

$$[\![d_1; \ldots; d_n]\!]_{typing} = [\![d_1]\!]_{typing} \circ_{\&} \ldots \circ_{\&} [\![d_n]\!]_{typing}$$

$$[\![v_1,\ldots,v_n : \tau]\!]_{typing} = [\![v_1,\tau]\!]_{typing-v} \circ_{\&} \ldots \circ_{\&} [\![v_n,\tau]\!]_{typing-v}$$

$$[\![v_1,\ldots,v_n : \tau \textbf{ when } C]\!]_{typing} = [\![v_1,\tau]\!]_{typing-v} \circ_{\&} \ldots \circ_{\&} [\![v_n,\tau]\!]_{typing-v}$$

$$[\![v,\tau]\!]_{typing-v} = \epsilon \text{ if } v \notin svar$$

$$[\![v,\tau]\!]_{typing-v} = [\![v]\!]_{var} \in [\![\tau]\!]_{type} \text{ if } v \in svar$$

$$[\![\textbf{int}]\!]_{type} = LUSTRE\_INT$$

$$[\![\textbf{bool}]\!]_{type} = LUSTRE\_BOOL$$

Instead of using `LUSTRE_BOOL` and `LUSTRE_INT`, it would also be possible to type the *state variables* using $\mathbb{P}(REF \times \tau)$ only. But the decision has been made to use `LUSTRE_BOOL` and `LUSTRE_INT` so that the translation to B itself is verified.

7.6.2 Nil Checking of Output Variables

As mentioned before, output parameters of the main node must never be `nil` during the program's execution. So, it must be ensured that the main node's output values are never equal to ∅ once its clock has been active. The predicate for generating nil checking of output variables expects the output declarations AST node.

$$[\![d_1;\ldots;d_n]\!]_{not-nil} = [\![d_1]\!]_{not-nil} \;\&\; \ldots \;\&\; [\![d_n]\!]_{not-nil}$$

$$[\![v_1,\ldots,v_n : \tau]\!]_{not-nil} = \begin{cases} [\![\$basic]\!]_{i-status} = \textbf{FALSE} \Rightarrow [\![v_1]\!]_{var} \neq \varnothing \\ \&\ldots\;\& \\ [\![\$basic]\!]_{i-status} = \textbf{FALSE} \Rightarrow [\![v_n]\!]_{var} \neq \varnothing \end{cases}$$

$$[\![v_1,\ldots,v_n : \tau \textbf{ when } C]\!]_{not-nil} = \begin{cases} [\![C]\!]_{i-status} = \textbf{FALSE} \Rightarrow [\![v_1]\!]_{var} \neq \varnothing \\ \&\ldots\;\& \\ [\![C]\!]_{i-status} = \textbf{FALSE} \Rightarrow [\![v_n]\!]_{var} \neq \varnothing \end{cases}$$

Note that it is not necessary to check whether an output variable is a *state variable* as each output variable is also a *state variable* (see Section 5.4.1).

7.6.3 Safety Properties

Verification nodes always have a single output parameter of the type boolean. There, all safety properties are conjuncted and assigned to the variable that is returned from the node. As explained in Section 2.4.6, the idea is to verify that the output value is always `true` under the assumption that the assertions are `true`. The LUSTRE nodes are translated such that each reachable state fulfils all assertions. This is achieved by using the `SELECT` substitution as explained later.

$$[\![v : \textbf{bool}]\!]_{safety} = [\![\$basic]\!]_{i-status} = \textbf{FALSE} \Rightarrow [\![v]\!]_{var} = \{ref \mapsto \textbf{TRUE}\}$$

$$[\![v : \textbf{bool when } C]\!]_{safety} = [\![C]\!]_{i-status} = \textbf{FALSE} \Rightarrow [\![v]\!]_{var} = \{ref \mapsto \textbf{TRUE}\}$$

7.7 Generation of INITIALISATION Clause

The `INITIALISATION` clause is generated assigning the initial value of all variables that are declared in `VARIABLES`. As mentioned before, these are *state variables*, the variables describing the clocks' initialization statuses, and a variable for assertions named `assert_ok`. The variable `assert_ok` is initially assigned to `TRUE`. *State variables* are initially assigned to ∅ representing `nil`. Note that the initialization does not perform the first cycle. In contrast, the initialized variables represent the program's state before executing the first clock step. Each clock's initialization status is initially assigned to `TRUE`.

$$init = \begin{cases} assert\_ok := \textbf{TRUE}; \\ [\![c_1]\!]_{i-status} := \textbf{TRUE}; \\ \ldots; \\ [\![c_n]\!]_{i-status} := \textbf{TRUE}; & \text{if} \quad \begin{array}{l} clocks = c_1, \ldots, c_n, \\ svar = v_1, \ldots, v_n \end{array} \\ [\![v_1]\!]_{var} := \varnothing; \\ \ldots; \\ [\![v_n]\!]_{var} := \varnothing \end{cases}$$

7.8 Generation of OPERATIONS Clause

Finally, the `OPERATIONS` clause is generated containing an operation executing a clock step. The input parameters of this operation are generated from the corresponding declarations AST node. Furthermore, the top-level substitution is the `PRE` substitution, in which the input parameters are typed.
Again, the top-level substitution within the `PRE` substitution is a `VAR` substitution if it is necessary to introduce local variables. Local variables are composed of *non-state variables* and variables storing the previous values of other variables. The latter are variables that are used within `pre` operators. Note that the operand of a `pre` expression is always a variable that is part of the program's state as described in Section 5.3.5. Whether a variable is introduced to store the previous value of another variable depends on the locations of the `pre` expressions where the other variable is used in. Such a local variable is introduced if there exists at least one `pre` expression where the other variable is used in, which is located after the equation defining this variable in the optimized AST.
There are substitutions generated which assign these variables to the corresponding variables storing the previous value before translating the node body.
In the case that no local variables are introduced, it is not necessary to generate substitutions assigning the values of these variables storing previous values of the other variables.
Each input variable is also either declared in the `VAR` substitution or the `VARIABLES` clause depending on whether it is a *state variable*. It is assigned to the corresponding input parameter of the *clock step operation* under the condition that its clock is active in the current cycle. So, there are substitutions assigning input variables generated at the beginning of the operation. This is explained later in this section as well.
Performing a *clock step operation* in LUSTRE affects the node's output values. Furthermore, it is possible to invoke other nodes within an expression. This would lead to the behavior that the node is simulated first before the output values are returned. Setting the preference `ALLOW_OPERATION_CALLS_IN_EXPRESSIONS` in PROB would make it possible to use operation calls in expressions if they do not change the program's state. But as they might change the program's state, a *clock step operation* is translated without return parameters. Instead, the translation of equations and assertions ensures that node call instances are simulated first. Afterwards, the output values are read from the invoked

node instances. Note that each output variable is a *state variable* and is thus declared in the `VARIABLES` clause. That is why an output value can be read from the invoked node instance.
Despite assignments for input variables and variables for previous values, the body of the generated *clock step operation* contains substitutions translated from the node body, substitutions for cutting off unreachable states, and substitutions updating each clock's initialization status.

$$[\![input, body]\!]_{clock-step} = \begin{cases} \begin{array}{l} clock\_step([\![input]\!]_{params}) := \\ \quad \textbf{PRE} \\ \quad\quad [\![input]\!]_{params-typing} \\ \quad \textbf{THEN} \\ \quad\quad [\![input]\!]_{params-assign}\circ; \\ \quad\quad [\![body]\!]_{tr}\circ; \\ \quad\quad [\![body]\!]_{assert\_cut}\circ; \\ \quad\quad [\![c_1, \ldots, c_n]\!]_{clocks-update} \\ \quad \textbf{END} \end{array} & \text{if } \begin{array}{l} [\![body]\!]_{op-locals} = \epsilon, \\ clocks = c_1 \ldots, c_n \end{array} \end{cases}$$

$$[\![input, body]\!]_{clock-step} = \begin{cases} \begin{array}{l} clock\_step([\![input]\!]_{params}) := \\ \quad \textbf{PRE} \\ \quad\quad [\![input]\!]_{params-typing} \\ \quad \textbf{THEN} \\ \quad\quad \textbf{VAR } [\![body]\!]_{op-locals} \textbf{ IN} \\ \quad\quad\quad [\![input]\!]_{params-assign}\circ; \\ \quad\quad\quad [\![body]\!]_{h-asn}\circ; \\ \quad\quad\quad [\![body]\!]_{tr}\circ; \\ \quad\quad\quad [\![body]\!]_{assert\_cut}\circ; \\ \quad\quad\quad [\![c_1, \ldots, c_n]\!]_{clocks-update} \\ \quad\quad \textbf{END} \\ \quad \textbf{END} \end{array} & \text{if } \begin{array}{l} [\![body]\!]_{op-locals} \neq \epsilon, \\ clocks = c_1 \ldots, c_n \end{array} \end{cases}$$

7.8.1 Generation of Parameters

As explained before, the input parameters of the *clock step operation* are generated from the corresponding declarations AST node.

$$[\![d_1; \ldots; d_n]\!]_{params} = [\![d_1]\!]_{params}, \ldots, [\![d_n]\!]_{params}$$
$$[\![v_1, \ldots, v_n : \tau]\!]_{params} = [\![v_1]\!]_{param}, \ldots, [\![v_n]\!]_{param}$$
$$[\![v_1, \ldots, v_n : \tau \textbf{ when } C]\!]_{params} = [\![v_1]\!]_{param}, \ldots, [\![v_n]\!]_{param}$$

Each parameter is named starting with the prefix `param_`. Similar to the machine name and the variable names, this is done to avoid identifiers starting with an *underscore*. While LUSTRE identifiers might begin with an *underscore*, those of B do not.

$$[\![v]\!]_{param} = param \text{ if } param = param\_var\_to\_b(v)$$
$$param\_var\_to\_b(var) = format(\text{“}param\_\%s\text{“}, var)$$

The typing predicates for parameters are generated similar to those in the `INVARIANT` with slight differences. On the one hand, the input parameters' names differ from the *state variables* that are typed in the `INVARIANT`. On the other hand, the generated typing predicates for main nodes do not accept `nil` values as parameter. The representation for LUSTRE booleans and integers are represented by `LUSTRE_BOOL` and `LUSTRE_INT` respectively. In contrast, those that do not accept `nil` values are represented by `LUSTRE_BOOL_NOT_NIL` and `LUSTRE_INT_NOT_NIL`.

$$\llbracket d_1; \ldots ; d_n \rrbracket_{params-typing} = \llbracket d_1 \rrbracket_{params-typing} \ \& \ \ldots \ \& \ \llbracket d_n \rrbracket_{params-typing}$$
$$\llbracket v_1, \ldots , v_n : \tau \rrbracket_{params-typing} = \llbracket v_1, \tau \rrbracket_{param-typing} \ \& \ \ldots \ \& \ \llbracket v_n, \tau \rrbracket_{param-typing}$$
$$\llbracket v_1, \ldots , v_n : \tau \ \textbf{when} \ C \rrbracket_{params-typing} = \llbracket v_1, \tau \rrbracket_{param-typing} \ \& \ \ldots \ \& \ \llbracket v_n, \tau \rrbracket_{param-typing}$$
$$\llbracket v, \tau \rrbracket_{param-typing} = \llbracket v \rrbracket_{param} \in \llbracket \tau \rrbracket_{param-type}$$
$$\llbracket \textbf{int} \rrbracket_{param-type} = LUSTRE\_INT\_NOT\_NIL \text{ if } \textit{main\_node}(\textit{node\_name})$$
$$\llbracket \textbf{int} \rrbracket_{param-type} = LUSTRE\_INT \text{ if } \textit{not}(\textit{main\_node}(\textit{node\_name}))$$
$$\llbracket \textbf{bool} \rrbracket_{param-type} = LUSTRE\_BOOL\_NOT\_NIL \text{ if } \textit{main\_node}(\textit{node\_name})$$
$$\llbracket \textbf{bool} \rrbracket_{param-type} = LUSTRE\_BOOL \text{ if } \textit{not}(\textit{main\_node}(\textit{node\_name}))$$

As mentioned before, each input variable is either declared in the `VAR` substitution or the `VARIABLES` clause. This is a different variable than the one that is passed to the operation.
An input parameter might be sampled on a clock other than the *basic clock*. It is then possible that this clock is not active in the current cycle. In this case, it is known that the variable is declared in the `VARIABLES` clause as a result of Section 5.4.1.
Parameters that are sampled on an inactive clock must retain its old value. To achieve this, a substitution is generated for each input variable that assigns the corresponding variable in `VARIABLES` or `VAR` to the input parameter's value that is passed through the operation. This is done with the condition that its clock is active in the current cycle. The generation of these substitutions is done using the declarations AST node to access clock information.

$$\llbracket d_1; \ldots ; d_n \rrbracket_{params-assign} = \llbracket d_1 \rrbracket_{params-assign}; \ldots ; \llbracket d_n \rrbracket_{params-assign}$$
$$\llbracket v_1, \ldots , v_n : \tau \rrbracket_{params-assign} = \llbracket v_1 \rrbracket_{param-assign}; \ldots ; \llbracket v_n \rrbracket_{param-assign}$$
$$\llbracket v_1, \ldots , v_n : \tau \ \textbf{when} \ C \rrbracket_{params-assign} = \llbracket v_1 \rrbracket_{param-assign}; \ldots ; \llbracket v_n \rrbracket_{param-assign}$$

For an input parameter being sampled on the *basic clock*, an assignment is generated where the variable is assigned to the corresponding parameter. In contrast, for input parameters that are not sampled on the *basic clock*, an `IF` substitution is generated containing this assignment. The additional condition checks whether the clock is active in the current cycle.
Clocks that are passed as parameters through a node might be defined as `nil`. This could be the case if the invoking node does not use a variable as a clock, but the invoked node does. So clocks must be checked for `nil` values when generating assignments for parameters. This is modeled as a `PRE` substitution after the clock is assigned.

$$\llbracket v \rrbracket_{param-assign} = \llbracket v \rrbracket_{var} := \llbracket v \rrbracket_{param} \text{ if } \textit{clock}(v) = \$basic, v \notin clocks$$
$$\llbracket v \rrbracket_{param-assign} = \left\{ \begin{array}{l} \llbracket v \rrbracket_{var} := \llbracket v \rrbracket_{param}; \\ \textbf{PRE} \ \llbracket v \rrbracket_{var} \neq \varnothing \ \textbf{THEN} \ skip \ \textbf{END} \end{array} \right. \text{ if } \textit{clock}(v) = \$basic, v \in clocks$$

$$[\![v]\!]_{param-assign} = \begin{cases} \textbf{IF}[\![c]\!]_{active}\textbf{THEN} \\ \quad [\![v]\!]_{var} := [\![v]\!]_{param} \\ \textbf{END} \end{cases} \text{if } clock(v) \neq \$basic, v \notin clocks$$

$$[\![v]\!]_{param-assign} = \begin{cases} \textbf{IF}[\![c]\!]_{active}\textbf{THEN} \\ \quad [\![v]\!]_{var} := [\![v]\!]_{param}; \\ \quad \textbf{PRE } [\![v]\!]_{var} \neq \varnothing \textbf{ THEN } skip \textbf{ END} \\ \textbf{END} \end{cases} \text{if } clock(v) \neq \$basic, v \in clocks$$

Even though it is obvious for input variables sampled on the *basic clock* that they always accept the value that is passed as an input parameter, they are also assigned to them. This makes it possible to get rid of the input parameters that are passed through the operation in the translation of the node body. Otherwise, it would be necessary to check for each identifier expression whether it is an input parameter and on which clock it is sampled on to identify its name.

7.8.2 Generation of Local Variables

Local variables are introduced in the `VAR` substitution and consists of *non-state variables* and variables storing previous values of other variables.

$$[\![body]\!]_{op-locals} = [\![body]\!]_{h-vars} \circ, [\![v_1]\!]_{var} \circ, \ldots \circ, [\![v_n]\!]_{var} \text{ if } non - svar = v_1, \ldots, v_n$$

Each LUSTRE variables is either passed as a parameter or defined by exactly one equation. Furthermore, each *non-state variable* that is not sampled on the *basic clock* is assigned to $\varnothing$ as later explained in the translation of equations. This ensures *non-state variables* to be assigned and their types to be inferred. Variables storing previous values of other variables are introduced for those that are operand of a `pre` expression in the node body. Additionally, the variable it corresponds to must be used in at least one `pre` operator which is located after the equation defining this variable in the optimized AST. Appendix D.7.2 shows the complete inference rules for the generation of local variables and the assignments for variables storing previous values.

$$[\![s_1; \ldots; s_n;]\!]_{h-vars} = [\![s_1;, l_1]\!]_{h-vars} \circ, \ldots \circ, [\![s_n;, l_n]\!]_{h-vars}$$
$$\text{if } l_1 = \varnothing, l_2 = l_1 \cup def_{vars}(s_1;), \ldots, l_n = l_{n-1} \cup def_{vars}(s_{n-1};)$$

$$[\![v_1, \ldots, v_n = E;, dvars]\!]_{h-vars} = [\![E, dvars]\!]_{h-vars}$$

$$[\![\textbf{assert } E;, dvars]\!]_{h-vars} = [\![E, dvars]\!]_{h-vars}$$

$$[\![op\ E, dvars]\!]_{h-vars} = \epsilon \text{ if } op = \textbf{pre}, E \notin dvars$$

$$[\![op\ E, dvars]\!]_{h-vars} = [\![E]\!]_{h-var} \text{ if } op = \textbf{pre}, E \in dvars$$

As mentioned before, each introduced variable storing a previous value of another variable is assigned before translating the node body. There, each variable is assigned to the respective variable it corresponds to. As a side effect, it is ensured that the types of these variables are inferred as well. Note that variables within `pre` expressions are *state variables* as a result of Section 5.4.1.

$$\llbracket s_1; \ldots; s_n; \rrbracket_{h-asn} = \llbracket s_1;, l_1 \rrbracket_{h-asn} \circ_; \ldots \circ_; \llbracket s_n;, l_n \rrbracket_{h-asn}$$
$$\text{if } l_1 = \varnothing, l_2 = l_1 \cup \mathit{def}_{vars}(s_1;), \ldots, l_n = l_{n-1} \cup \mathit{def}_{vars}(s_{n-1};)$$

$$\llbracket v_1, \ldots, v_n = E;, dvars \rrbracket_{h-asn} = \llbracket E, dvars \rrbracket_{h-asn}$$

$$\llbracket \textbf{assert } E;, dvars \rrbracket_{h-asn} = \llbracket E, dvars \rrbracket_{h-asn}$$

$$\llbracket op\ E, dvars \rrbracket_{h-asn} = \epsilon \text{ if } op = \textbf{pre}, E \notin dvars$$

$$\llbracket op\ E, dvars \rrbracket_{h-asn} = \llbracket E \rrbracket_{h-var} := \llbracket E \rrbracket_{var} \text{ if } op = \textbf{pre}, E \in dvars$$

These variables start with the prefix hvar_ to avoid identifier names starting with an *underscore* as they are not allowed in B.

$$\llbracket v \rrbracket_{h-var} = var \text{ if } var = hist\_var(v)$$
$$hist\_var(var) = \ format(\text{“}hvar\_\%s\text{“}, var)$$

7.8.3 Translation of Node Body

Equations, assertions, and expressions make up the node body. The translation of these constructs will now be explained. As a result of Section 5.3.5, the AST that is passed to LUSTRE2B is rewritten such that sequential B code can be generated for the node body. So, the translation results in the sequential composition of the translation of the equations and assertions.
During the translation of each equation, assertion, and expression, a list of variables that are defined in previous equations is taken into account. It is used for the translation of pre expressions which is explained later.

$$\llbracket s_1; \ldots; s_n; \rrbracket_{tr} = \llbracket s_1;, l_1 \rrbracket_{tr}; \ldots; \llbracket s_n;, l_n \rrbracket_{tr}$$
$$\text{if } l_1 = \varnothing, l_2 = l_1 \cup \mathit{def}_{vars}(s_1;), \ldots, l_n = l_{n-1} \cup \mathit{def}_{vars}(s_{n-1};)$$

Equations Consider an equation $v_1, \ldots, v_n = E$. When translating this equation, operation calls are generated for node calls in E first. Applying an operation call means that a *clock step* is executed for the node instance it corresponds to. Afterwards, its updated return values are read when translating the expression E.
The translation of E results in a single expression if it is a node call returning a single value or any other expression. This is due to the reason that expression lists are rewritten in the AST that is passed to LUSTRE2B as explained in Section 5.3.5. In contrast, if E is a node call that returns more than one value, then the translation of this expression yields a list of B expressions.
Let v_i be i-th variable on the left-hand side and e_i be the corresponding i-th translated expression of E. Each of the v_i together with its corresponding e_i is used to generate a substitution.

$$\llbracket v_1, \ldots, v_n = E;, dvars \rrbracket_{tr} = \begin{cases} \llbracket E, dvars \rrbracket_{node-sim} \circ_; \\ \llbracket v_1, e_1 \rrbracket_{eq}; \\ \ldots; \\ \llbracket v_n, e_n \rrbracket_{eq}; \end{cases} \text{if } \llbracket E, dvars \rrbracket_{tr} = e_1, \ldots, e_n$$

This substitution assigns the translation of v_i as variable to e_i under the condition v_i's clock is active. This means that an assignment is generated where v_i is assigned to e_i. If

the corresponding clock is not the *basic clock*, then an `IF` substitution is generated containing this assignment. There it is checked whether the clock is active. Note that clocks, and variables that are sampled on a clock other than the *basic clock* are always *state variables*. For defined variables that are sampled on the *basic clock*, it is not necessary to generate an `IF` substitution as the *basic clock* is always active.
With the list of clocks that is passed to LUSTRE2B, it is possible to determine whether v_i is a clock. In this case, it is necessary to generate a `PRE` substitution within a sequential composition after the generated assignment. This substitution checks whether the clock is intended to be assigned to `nil`. This leads to the *abort state* as explained before.

$$[\![v, e]\!]_{eq} = [\![v]\!]_{var} := e \text{ if } clock(v) = \$basic, v \notin clocks$$

$$[\![v, e]\!]_{eq} = \begin{cases} \textbf{IF } [\![c]\!]_{active} \textbf{ THEN} \\ \quad [\![v]\!]_{var} := e & \text{if } clock(v) \neq \$basic, v \notin clocks \\ \textbf{END} \end{cases}$$

$$[\![v, e]\!]_{eq} = \begin{cases} [\![v]\!]_{var} := e; \\ \textbf{PRE } [\![v]\!]_{var} \neq \varnothing \textbf{ THEN } skip \textbf{ END} \end{cases} \text{if } clock(v) = \$basic, v \in clocks$$

$$[\![v, e]\!]_{eq} = \begin{cases} \textbf{IF}[\![c]\!]_{active}\textbf{THEN} \\ \quad [\![v]\!]_{var} := e; \\ \quad \textbf{PRE } [\![v]\!]_{var} \neq \varnothing \textbf{ THEN } skip \textbf{ END} \\ \textbf{END} \end{cases} \text{if } clock(v) \neq \$basic, v \in clocks$$

Assertions For an assertion `assert E`, simulation of node call instances in E are generated first. This is also required to access the updated output values of the node call instances in E. Despite the node call simulations, two substitutions are generated for an assertion. Similar to output parameters and clocks, assertion expressions must never be `nil`. This is modeled as an *abort state* as this leads to undefined behavior. So, the first one checks that the translated expression is unequal $\varnothing$ using a `PRE` substitution. In contrast, the second substitution checks whether the expression is `true` in LUSTRE values' representation i.e. $\{ref \mapsto \textbf{TRUE}\}$. The result is then conjuncted with `assert_ok` and stored as the new value of this variable.

$$[\![\textbf{assert } E; , dvars]\!]_{tr} = \begin{cases} [\![E, dvars]\!]_{node-sim} \circ; \\ \textbf{PRE } [\![E, dvars]\!]_{tr} \neq \varnothing \textbf{ THEN } skip \textbf{ END}; \\ assert\_ok := bool(assert\_ok = \textbf{TRUE} \ \& \ [\![E, dvars]\!]_{tr} = \{ref \mapsto \textbf{TRUE}\}) \end{cases}$$

Assertions in LUSTRE are assumptions about the program. So, this is realized using a `SELECT` substitution to cut off states where all assertion expressions are `false`. Within the node body, assertion expressions are only checked for `nil` modeling an *abort*, while states are not cut off immediately. Cutting off states using `SELECT` is done after the node body's translation. There, the value of `assert_ok` in the current machine and all included machines are checked to be `true`. The introduction of the variable `assert_ok` makes it possible to check all assertion expressions for `nil`. If the `SELECT` substitution is generated instead of the assignment of `assert_ok`, then states containing `nil` in assertions might have been cut off. This is also the reason why such a `SELECT` substitution is generated for the main node only. The complete translation rules for the `SELECT`'s condition are shown in Appendix D.7.3. There, the node body is visited recursively to access each included machine's `assert_ok` variable.

$$[\![body]\!]_{assert-cut} = \textbf{SELECT } [\![body]\!]_{assert-true} \textbf{ THEN } skip \textbf{ END} \quad \text{if } main\_node(node\_name)$$

$$[\![body]\!]_{assert-cut} = \epsilon \quad \text{if } not(main\_node(node\_name))$$

$$[\![s_1; \ldots; s_n;]\!]_{assert-true} = assert\_ok \ = \ \textbf{TRUE} \ \circ_{\&} \ [\![s_1;]\!]_{assert-true} \circ_{\&} \ldots \circ_{\&} [\![s_n;]\!]_{assert-true}$$

$$[\![node(p_1, \ldots; p_n)]\!]_{assert-true} = \begin{cases} [\![p_1, \ldots, p_n]\!]_{assert-true} \circ_{\&} \\ iname.assert\_ok \ = \ \textbf{TRUE} \end{cases} \text{if } \begin{array}{l} nid = id(node(p_1, \ldots, p_n)), \\ iname = instance\_name(nid) \end{array}$$

Node Call Simulation When translating equations and assertions, node call instances within them are simulated first. To generate substitutions for simulation of node call instances, each equation and assertion AST node is visited recursively. The complete translation rules for node call simulations are portrayed in Appendix D.7.3.
Each time a node call is detected, it is first inspected for other node calls within its parameters. Similar to equations and assertions, node call instances within the parameters are simulated first. This makes it possible to access the return values of the node calls within the parameters of a node call.
Again, the machine instance representing the node call is simulated under the condition that its *basic clock* is active in the invoking node. Clocks are always declared before they are used. So, the *basic clock* of a node is always the clock of its first parameter. As the *basic clock* is always active, only an operation call invoking the *clock step operation* of the corresponding machine instance is generated. For clocks other than the *basic clock*, this operation call is within the body of an `IF` substitution which checks whether the node call's clock is active. The operation call is performed using the translation of the expressions in the parameters.

$$[\![node(p_1, \ldots, p_n), dvars]\!]_{node-sim} = \begin{cases} [\![(p_1, \ldots, p_n), dvars]\!]_{node-sim} \circ_; \\ [\![nid]\!]_{clock-step-name}([\![(p_1, \ldots, p_n), dvars]\!]_{tr}) \end{cases}$$
if $nid = id(node(p_1, \ldots, p_n)), \ clock(p_1) = \$basic$

$$[\![node(p_1, \ldots, p_n), dvars]\!]_{node-sim} = \begin{cases} [\![(p_1, \ldots, p_n), dvars]\!]_{node-sim} \circ_; \\ \textbf{IF } [\![c]\!]_{active} \textbf{THEN} \\ \quad [\![nid]\!]_{clock-step-name}([\![(p_1, \ldots, p_n), dvars]\!]_{tr}) \\ \textbf{END} \end{cases}$$
if $nid = id(node(p_1, \ldots, p_n)), \ clock(p_1) = c, c \neq \$basic$

$$[\![nid]\!]_{clock-step-name} = iname.clock\_step \text{ if } iname = instance\_name(nid)$$

Data Expressions Data expressions in LUSTRE are literals, identifier expressions, arithmetic expressions, logical expressions, comparisons, and if-then-else expressions. Literals are integers, booleans, and real numbers. As real numbers are not supported in B yet, they are not supported in the translation to B.

As explained before, LUSTRE values are generated as relations. This is the way how the implemented operators in `LibraryLustre` can be used for them.

$$[\![lit, dvars]\!]_{tr} = \{ref \mapsto lit\} \text{ if } literal(lit)$$

Implementing the operators in `LibraryLustre` also provides the opportunity to use the variables without accessing its values in the generated B code. They are translated making use of the *state variables* and *non-state variables* declared in `VARIABLES` and `VAR` respectively.

$$[\![v, dvars]\!]_{tr} = [\![v]\!]_{var} \text{ if } variable(v)$$

Arithmetic expressions, logical expressions, comparisons, and if-then-else expressions are translated using the operators in `LibraryLustre`. With this approach, the operands are translated and used as operands in the invocation of the corresponding implemented operator. All translation rules are portrayed in Appendix D.7.3.

Temporal Expressions Temporal operators that are supported in this work are the operators `->`, `pre`, `when`, and `current`.
To define the first value of an expression flow, the operator `->` is used. As described in Section 5.3.5, the rewritten AST avoids nested `pre` expressions. In general, temporal expressions using the operators `pre`, `when`, and `current` are rewritten such that its operands are always stored in variables.

By following this approach, `->` expressions can be treated like `if-then-else` expressions with the condition being its clock's initialization status. The corresponding operator is implemented in `LibraryLustre` for both supported types.

$$[\![E_1 \texttt{ -> } E_2, dvars]\!]_{tr} = \mathit{l\_fby\_integer}([\![C]\!]_{i-status} \mapsto [\![E_1, dvars]\!]_{tr} \mapsto [\![E_2, dvars]\!]_{tr}) \\ \text{if } \mathit{clock}(E_1 \texttt{ -> } E_2) = C, \tau(E_1) = \tau(E_2) = \textbf{int}$$

$$[\![E_1 \texttt{ -> } E_2, dvars]\!]_{tr} = \mathit{l\_fby\_boolean}([\![C]\!]_{i-status} \mapsto [\![E_1, dvars]\!]_{tr} \mapsto [\![E_2, dvars]\!]_{tr}) \\ \text{if } \mathit{clock}(E_1 \texttt{ -> } E_2) = C, \tau(E_1) = \tau(E_2) = \textbf{bool}$$

Let $pre(E)$ be an expression, then its operand E is always a variable after optimizing the AST. The translation of E yields the corresponding *state variable* if it was not defined in a previous equation in the optimized AST. Note that the variable representing the operand of `pre` is always stored in the program's state. Otherwise, it is translated as the variable storing the previous value of E that is introduced in the `VAR` substitution. To achieve this, a list of variables that are defined in previous equations is passed to the predicates for translating equations, assertions, and expressions.

$$[\![\textbf{pre } E, dvars]\!]_{tr} = [\![E]\!]_{var} \text{ if } E \notin dvars$$

$$[\![\textbf{pre } E, dvars]\!]_{tr} = [\![E]\!]_{h-var} \text{ if } E \in dvars$$

This makes it possible to reduce the number of introduced variables in the `VAR` substitution. Consider a variable that is used several times in the `pre` operator with the corresponding `pre` expressions always being located before the equation defining this variable. It is then not necessary to declare a variable storing its previous value in the `VAR` substitution.
The `when` operator samples an expression on a slower clock. Similar to `pre`, its operands are always variables after optimizing the AST. Note that each clock must be a variable in the supported subset of LUSTRE.

Similar to the discussion in Section 6.3.3, the left-hand side operand is only used in the context where the clock on the right-hand side is active. That is why it is possible to translate a `when` expression to the left-hand side operand.

$$[\![E \textbf{ when } C, dvars]\!]_{tr} = [\![E]\!]_{var}$$

Unlike the `when` operator, the `current` operator is used to sample an expression on a faster clock in the hierarchy. Similar to `pre` and `when` the operand of a `current` operator is always a variable. The translation aims to translate a `current` expression such that it holds the value of the operand the last time it was active. As this operand is a variable that was updated the last time its clock is active, each `current` expression is translated to this variable. Note that the variable representing the operand of `current` is always stored in the program's state as it is sampled on a clock slower than the *basic clock*.

$$[\![\textbf{current } E, dvars]\!]_{tr} = [\![E]\!]_{var}$$

Node Call Expression As explained before, machine instances representing node calls are simulated before they are used as expressions. This provides the opportunity to access the updated return values.
When translating a node call as an expression, it is translated to a list of expressions. Each expression within this list is translated such that it accesses a return value of the corresponding node. So, the output values are read from the machine instance representing the node call.

$$\begin{array}{l} [\![node(p_1, \ldots, p_n), dvars]\!]_{tr} = [\![o_1, nid]\!]_{read-out}, \ldots, [\![o_m, nid]\!]_{read-out} \\ \text{if } id(node(p_1, \ldots, p_n)) = nid,\ output(node(p_1, \ldots, p_n)) = o_1, \ldots, o_m \end{array}$$

$$[\![o, nid]\!]_{read-out} = iname.[\![o]\!]_{var} \text{ if } iname = instance\_name(nid)$$

7.8.4 Generation of Clock Status Update

Finally, substitutions updating each clock's initialization status are generated. Note that each of these variables describes whether the next cycle is the clock's first cycle. Thus, the updated values are used for the calculation of the next cycle. For each clock, its value is set to `FALSE` if the clock is active in the current cycle. This is realized with an `IF` substitution for clocks other than the *basic clock*. For the *basic clock*, it is obvious that this clock is always active.

$$[\![c_1, \ldots, c_n]\!]_{clocks-update} = [\![c_1]\!]_{clock-update}; \cdots ; [\![c_n]\!]_{clock-update}$$

$$[\![c]\!]_{clock-update} = [\![c]\!]_{i-status} := \textbf{FALSE} \text{ if } c = \$basic$$

$$[\![c]\!]_{clock-update} = \begin{cases} \textbf{IF } [\![c]\!]_{active} \textbf{ THEN} & \\ \quad [\![c]\!]_{i-status} := \textbf{FALSE} & \text{if } c \neq \$basic \\ \textbf{END} & \end{cases}$$

7.8.5 Clock Activeness

The predicate that is generated checking the activeness of a clock is a conjunction. There, the given clock and all its *super clocks* are checked whether they hold the value
`{ref` $\mapsto$ **`TRUE`**`}`. For the determination of the *super clocks*, the *clock hierarchy* is taken

into account. It does not contain the *basic clock* as it is obvious that the *basic clock* is always active and a *super clock* of other clocks. Note that all clocks other than the *basic clock* are *state variables*.

$$superclocks(c) = list(\{c' | c' \mapsto c : chierarchy\})$$

$$[\![c]\!]_{active} = \begin{cases} [\![c_1]\!]_{var} = \{ref \mapsto \textbf{TRUE}\} \ \& \\ \ldots \ \& \\ [\![c_n]\!]_{var} = \{ref \mapsto \textbf{TRUE}\} \ \& \\ [\![c]\!]_{var} = \{ref \mapsto \textbf{TRUE}\} \end{cases} \text{if } superclocks(c) = c_1, \ldots, c_n$$

8 Empirical Analysis of the Performance

In this section, the performance of simulation and model checking are investigated. For both tasks, the performance of the B translations will be compared to interpretation using the LUSTRE AST interpreter.
When exploring the performance of simulation, generated code from Java and C++ code using B2PROGRAM are also taken into account. As B2PROGRAM generates code from a verified B model, the generated code gets rid of constructs that are relevant for verification such as invariants and preconditions. That is why the performance of B2PROGRAM is not investigated in the context of a C++ or Java model checker. To achieve a fairer comparison with the PROB approaches, the simulation benchmarks are executed without invariant checking. The generated Java and C++ code use primitive integers, while both PROB approaches are implemented using big integers. This is an advantage of B2PROGRAM if it is ensured that there are no integer overflows in the given B model. The internal representation for sets and relations uses the Clojure and immer library for Java and C++ respectively.
Code generation from B2PROGRAM is applied with the Git version *9f19a539820fbcebd486aa8f2a6791879ac95afd*. Furthermore, the C++ benchmarks are compiled with the *clang compiler (Apple clang version 11.0.0 (clang-1100.0.33.8))* using both optimization options `-O1` and `-O2`. Again, the generated Java code run on the *Java Virtual Machine (OpenJDK 64-Bit Server VM (build 14.0.1+7, mixed mode, sharing))*. In contrast, the B translations and the interpretation using the LUSTRE AST interpreter are executed using PROB *(Git Hash: df48a6e3ae348791e30465250eb5ec7c4eb1c8ec)*. The LUSTRE AST interpreter is compiled together with PROB which leads to a speedup of a factor around five to six.
During the performance analysis of model checking, those of explicit-state model checking for both PROB approaches are taken into account. Bounded model checking (BMC) is also applied to the B models that are translated from LUSTRE programs.
To explore the performance of the LUSTRE constructs, microbenchmarks are considered. They include those for LUSTRE operators, assertions, clocks, and node calls. While the benchmarks of realistic LUSTRE nodes are more representative for the general practice of LUSTRE, microbenchmarks provide a more precise view to detect performance lacks. The results and the way how the benchmarks are exactly measured are described in the respective subsections. Furthermore, the runtime for the microbenchmarks and simulation of both PROB approaches also include the parsing time of the respective programs. However, the execution of the operations takes a long time such that the

parsing time is not relevant. All benchmarks are analyzed according to the runtime and the memory (maximum resident set size) and executed on a MacBook Air with 8 GB of RAM and a 1.6 GHz Intel i5 processor with two cores. For the benchmarks in B, the flag `COMPRESSION` is set to `TRUE`. This is also done for the flags `SMT` and `SYMBOLIC` in the BMC benchmarks.

8.1 Description of the Benchmarks

For each LUSTRE operator, a node is created executing the corresponding operator with the operands given as parameters and returning the result as its output. Each `current` microbenchmark also applies a `when` operation to fulfil the clock checking rules. Additionally, there are also microbenchmarks for the assertion, the node call, and clocks. The microbenchmarks for assertion and node call execute an `and` operation resulting in `true` which is again defined as the result of the respective assertion and the invoked node. In contrast, the clocks' benchmarks are separated into clocks that are passed as parameters and clocks that are defined within the node locally. Both consist of five clocks in addition to the *basic clock* and perform a various number of `current` and `when` expressions.
Despite the microbenchmarks, there are various LUSTRE programs chosen for the empirical analysis of the performance. These benchmarks range from small LUSTRE programs such as Lift and the U-Turn Section Management System (UMS) to larger programs such as Pilot Flying and Docking Approach.

Lift is a simple program controlling the movement of a lift with 100 floors. It cannot be moved up and down if it is located on the highest floor and the lowest floor respectively.
SLOW TIME STABLE is a program performing a stopwatch [15]. It contains parameters for setting the delay, the corresponding value, and a clock `second` on which the delay is sampled. Furthermore, it is implemented containing two nodes, a node invocation, and a `when` operation.
UMS is implemented managing the switch and the trains' movements in the U-Turn section of a subway line [17]. Its implementation makes use of boolean values and logical operations only.
Train Speed is a program measuring the speed of a train. Its task is detecting whether a train is early or late [28]. It is implemented using many (nested) `if-then-else` operations and a node invocation. Compared to Lift and UMS, it has a higher number of operations and is thus slightly more complex.
The benchmarked LUSTRE programs also include different components of a production cell controller. These are the LUSTRE nodes Moving Item, Press, Rbase, and Rgrips. [29] Furthermore, there is also a verification node for Moving Item which is named Verify Moving Item. On the one hand, these LUSTRE nodes are implemented containing boolean values only. On the other hand, they contain a high number of parameters and node calls. Most node calls are used implementing the program's temporal behavior.
UMS Verification [17] and Train Speed Verification [28] are the respective verification nodes of UMS and Train Speed. Both of them contain safety properties and node invocations. Additionally, UMS Verification is implemented with assertions. There are many safety properties in UMS Verification which have to be checked to ensure that no accidents can happen. Again, the safety properties in Train Speed Verification check that the train is never both early and late. Furthermore, the result cannot change from early to

late and the other way around immediately.
Another LUSTRE program that is considered in this work is Carlights. It expects signals for the switch position (`ON`, `OFF` or `AUTO` modeled as 0 to 2) and the intensity (0 to 10) [30]. This program is implemented using a clock for each switch position. Using all this information, it determines whether the car light is turned on in the current cycle. Compared to other programs, it is implemented containing many clocks and `when` operations. Outgoing from the initial version, it is compiled to a subset of LUSTRE that is supported by this work using the *Lustre V6 Compiler*[11].
The most complex programs are Pilot Flying, Submode Logic, and Docking Approach. All three programs are taken from [31]. Pilot Flying is a program handling the side-stick in an airplane's cockpit. Again, Docking Approach implements the docking of a space shuttle with the International Space Station (ISS). It contains several operations for orienting the space shuttle, docking with the ISS, and capturing the latch [10, 32]. Furthermore, Submode Logic is a LUSTRE node which implements the mode logic for a NASA project. Each of them is composed of a single node which consists of a high number of operations and variables. While most of the variables in Pilot Flying are of the type boolean, those of Docking Approach are mostly integers. Submode Logic is implemented using approximately as many integers as booleans. Furthermore, Pilot Flying is a program containing many `pre` and logical operations. In contrast, Submode Logic and Docking Approach are implemented consisting of many `if-then-else`, logical, and = operations.
Additionally, the performance of Heat Controller [30], Vehicle Speed [30], and Cruise Controller [31] are investigated as well. As all three programs are implemented using real numbers, they are not translated to B and are thus not translated to Java and C++ using B2PROGRAM.
Heat Controller is a program expecting the temperature from three sensors to determine whether the heat should be turned on. These sensors are also used to detect whether there are defect sensors. In the case that all of them are defect, the heat controller is restarted and its sensors are recalibrated. The temperature rises 0.5°C each clock step if the heat is turned on. Otherwise, it cools down by 0.5°C. Heat Controller contains many `if-then-else`, comparisons, and arithmetic operations.
Vehicle Speed contains two sensors: One for detecting the rotation of a vehicle's wheels and another one as a time sensor. Using both of them, it calculates the vehicle's speed. Compared to most other programs, it is implemented with many `when` operations and thus consists of a clock other than the *basic clock*. Similar to Carlights, Vehicle Speed is also compiled to a subset of LUSTRE that can be handled by this work.
Cruise Controller is a large program managing a car's speed automatically. It is composed of many `if-then-else`, comparison, and logical operations.

8.2 Microbenchmarks

As mentioned before, microbenchmarks are taken into account to analyze the LUSTRE constructs' performance. They are executed in the context of the LUSTRE AST interpreter, the translation to B, and the execution in Java and C++. Operations on real numbers are only analyzed using the LUSTRE AST interpreter as they are not supported in B.
The microbenchmarks are executed without invariant checking similar to the simulation

[11]https://gricad-gitlab.univ-grenoble-alpes.fr/verimag/synchrone/lustre-v6

benchmarks. Each of them is executed 500 000 times in a loop. This is done five times and afterwards, the median value is taken as the result. All results are portrayed in Appendix F.

The results show that interpreting the LUSTRE AST is significantly faster than the corresponding translation to B for all microbenchmarks. One can also see that the execution in Java leads to a slightly better runtime than the LUSTRE AST interpretation for the most microbenchmarks. Again, C++ seems to be faster than Java.
The compilers for both programming languages perform some optimizations. On the one hand, the JIT compiler in Java is possibly turned on for loops with many steps [33]. On the other hand, the execution in C++ could be optimized out at compilation. To make a better conclusion, benchmarks of more realistic machines are considered in this section. It could be the case that the *clang compiler* can no longer optimize that much or either that the Java JIT compiler is not turned on.
Representing LUSTRE expressions as relations in B possibly leads to the result that there are no differences between executing C++ code optimized with `-O1` and `-O2`. [7]

The results also show that operations on booleans are executed faster than on integers in the B translations. When one sets the flag `SYMBOLIC`, then the boolean microbenchmarks yield similar results like those for integers. It seems that operations on booleans are faster when the implemented functions are represented as enumerated sets rather than when they are represented symbolically. For data operators and `->`, the speedup is up to a factor of four. Note that `->` is treated similar to `if-then-else` with the difference that the condition of `if-then-else` is checked for `nil`. This is possibly the reason why `->` yields a slightly better runtime for both approaches. According to the LUSTRE AST interpreter, it seems that there are no differences between the evaluation of values with different types.
The other data operators (`pre`, `when`, and `current`) also seem to be applied on booleans slightly faster than on integers in the translation to B. There, the speedup is up to a factor less than two. The lower speedup could be explained that the translations mostly perform lookups and logical operations to check whether the clock is active.
Both clocks microbenchmarks are executed using the LUSTRE AST interpreter faster than execution in Java and C++. According to these two microbenchmarks, Java yields a faster runtime than C++. One can also see that the benchmark for parameter clocks yield a faster runtime for the LUSTRE AST interpreter, while local clocks seem to be treated faster for the translation to B as well as code generation to Java and C++. In general, it also seems that optimization in Java and C++ cannot be applied to these two benchmarks well. The result that Java is faster than C++ seems to be caused by the generated *if-statements* where the *clang compiler* possibly could not optimize well.
According to memory usage, one can see that the translation to B requires more memory than the other approaches. Especially the clocks' microbenchmarks for the translation to B and Java require more memory than the other microbenchmarks which comes to no surprise as this microbenchmark contains more variables. Furthermore, the memory usage for Java is higher than the LUSTRE AST interpreter and C++. Concerning Java, this could be caused by the variables' internal representation. They are represented as Clojure maps and seem to require more memory. As there are no set operations in the translation to B, it seems that no advantages can be gained from the structural sharing of the persistent data structures.

Comparing the assertion microbenchmark and the node call microbenchmark to the `and` operator, one can see that there is a greater impact on the runtime for the translations to B. For the LUSTRE AST interpreter, Java, and C++, the runtime is either approximately as fast or only affected slightly.

8.3 Simulation

The performance of simulation is analyzed for the LUSTRE AST interpreter, LUSTRE2B, and code generation targeting Java and C++ using B2PROGRAM. Compared to the microbenchmarks, more realistic programs are now taken into account. Invariant checking is also turned off for both PROB approaches. Each benchmark is executed five times and afterwards, the median value is taken for the result.
To explore the performance, a trace within the state space is chosen as a benchmark for each program. Most of the traces contain a cycle where a high number of clock step operations are executed in a loop. They should also cover a high number of parameter combinations such that it is representative for the benchmarked program. All clock step operations of a benchmark are performed by an XTL or B specification that includes the respective LUSTRE program or B model. E.g. the benchmark for `PilotFlying.lus` is realized by the XTL specification `PilotFlying.P` which initializes the program in the `start/1` predicate and executes all clock step operations in `trans/2`. The difference between this XTL specification and the loading file is that this one executes a high number of operations in `trans/2`. Again, the benchmark for `M_pilot_flying.mch` is realized by the B model `M_pilot_flying_exec` which includes `M_pilot_flying`. It contains an operation `simulate` which performs all clock step operations. The benchmarks for B, Java, and C++ are then executed by triggering `simulate`. For the LUSTRE AST interpreter, it is done by triggering a single operation of the XTL specification that performs all clock steps.
Lift moves the lift to the top level and the ground level 2000 times. Within this loop, the first command is always staying at the ground floor. So the loop contains 201 operations where the first operation never moves the lift. Again, SLOW TIME STABLE sets the timestamp to three and then counts it down to zero. This is repeated 100 000 times.
UMS simulates a complete process of a U-Turn for a subway 10 000 times. The process contains entering the critical section, performing a U-Turn, and finally leaving the section. To compare the performance of UMS and its verification node, UMS Verification also executes the same path 10 000 times.
The Train Speed benchmark executes all parameter combinations 20 times within a loop that runs for 1000 times. Both Train Speed and Train Speed Verification also perform the same actions such that both nodes can be compared with each other.
To investigate the performance of the production cell controller's nodes, all parameter combinations are executed once in a loop for each node. Moving Item and Verify Moving Item consists of eight parameter combinations. All of them are executed in a loop 10 000 times. Again, Press and Rbase are implemented having 512 possibilities to combine the parameters. For both programs, 100 clock step operations are applied in a loop. Rgrips is implemented with 2048 combinations of its parameters. Its clock step operations are executed 25 times. The generated clock step operation for Press, Rbase, and Rgrips in Java and Rgrips in C++ are too large to be handled by the respective compilers. So these

programs are modified manually to make the code executable.
Carlights performs all combinations of the switch position and the intensity controlling the car lights 1000 times in a loop. Clock step operations with all parameter combinations are applied for Submode Logic 1000 times as well. Again, Pilot Flying executes 201 operations in the side system of an airplaine's cockpit before 85 operations are executed in a loop for 100 times. Docking Approach performs a total number of 1099 operations until the space shuttle is docked with the ISS. The trace is chosen such that it performs a higher number of clock steps. It would also be possible to finish the process of docking in fewer steps. As later explained in this section, this trace is found with explicit-state model checking.
To analyze the performance of Heat Controller, it is warmed up from 14.5°C until 25.5°C. Afterwards, it is cooled down to 14.5°C again. This is applied 1500 times.
In contrast, the sensors of Vehicle Speed detects a timestamp and a rotation in the beginning. During the execution, there are eight rotations without a timestamp before a timestamp is recognized without a rotation. These commands are performed 1000 times in a cycle.
Cruise Controller simulates a process containing 800 commands in which it is turned on and off with different commands for accelerating and deaccelerating. It also contains commands for canceling, braking, and for invalid inputs which are commands that concern safety conditions. The 800 commands are executed 15 times in a loop.

The result of the runtime shows that simulation using LUSTRE AST interpreter is faster than the execution of the B models up to more than one magnitude. For all presented programs, the performance can be improved by code generation to Java and C++ using B2PROGRAM. While most microbenchmarks are executed in C++ faster than in Java, this is not always the case for more realistic programs. Sometimes the generated Java code is faster than C++, sometimes it is the other way around. For some benchmarks, the generated code is also approximately as fast for both programming languages.
On the one hand, this could be caused by the optimization options `-O1` and `-O2` which seem to be less effective for more complex expressions. On the other hand, one must take into account that the JVM activates the JIT compiler for programs with long-running loops which leads to better performance [33]. In general, it seems that C++ can handle larger LUSTRE programs and LUSTRE programs which are not executed in a long-running loop better than Java.

According to smaller LUSTRE programs, simulation using the LUSTRE AST interpreter is significantly faster than the translation to B. Comparing Lift, Train Speed, and SLOW TIME STABLE with UMS, one can see that programs with many data operations on integers lead to a higher speedup. This could be explained by the microbenchmarks of operations on integers and booleans. UMS Verification and Train Speed Verification are slower than the respective nodes they verify which comes to no surprise. Many safety properties are added to both verification nodes. While many assertions and node calls are added in UMS Verification, Train Speed Verification is implemented without an assertion and fewer node calls. This possibly leads to the behavior that Train Speed Verification is only slightly slower than Train Speed, while UMS Verification is significantly slower than UMS. One can also see that the speedup of the LUSTRE AST interpreter compared to the translation to B is significantly higher for UMS Verification than UMS.
While UMS and Train Speed are executed in C++ faster than in Java, for Lift, SLOW

Table 16: Runtimes of Smaller LUSTRE Programs According to LUSTRE AST interpretation, Translation to B, and Code Generation to Java and C++ in Seconds with Number of Clock Steps (CK Steps), Memory Usage in KB, PI = Primitive Integer

Lift		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
(402 000 CK steps)	Runtime	265.75	7.33	5.67	6.13	6.24
	Memory	1 344 584	167 652	308 136	852	864
	CK Step/s	1512	54 843	70 899	65 579	64 423
SLOW TIME STABLE		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
(600 000 CK steps)	Runtime	246.49	11.77	5.90	6.27	6.33
	Memory	2 103 384	167 468	295 280	824	832
	CK Step/s	2434	50 977	101 694	95 693	94 786
UMS		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
(60 000 CK steps)	Runtime	10.68	3.05	2.39	1.09	1.13
	Memory	406 412	167 492	171 720	796	808
	CK Step/s	5617	19 672	25 104	55 045	53 097
Train Speed		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
(80 000 CK steps)	Runtime	107.81	4.28	3.67	3.14	3.09
	Memory	406 940	167 440	309 116	852	864
	CK Step/s	742	18 691	21 798	25 477	25 889
UMS Verification		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
(60 000 CK steps)	Runtime	665.05	22.59	6.74	7.82	7.86
	Memory	2 105 168	167 488	338 096	904	916
	CK Step/s	90	2656	8902	7672	7633
Train Speed Verification		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
(80 000 CK steps)	Runtime	123.21	5.63	3.90	3.95	3.96
	Memory	585 740	167 452	303 112	852	864
	CK Step/s	649	14 209	20 512	20 253	20 202

TIME STABLE, and UMS Verification, it is the other way around. Again, Train Speed Verification is executed in Java as fast as in C++. Especially for Lift and SLOW TIME STABLE, it seems that the Java JIT compiler handles the long-running loop better than *clang* compiler's optimization. For LUSTRE programs consisting of many nodes e.g. UMS Verification, it seems that the *clang* compiler cannot optimize well, too.

Table 17: Runtimes of Production Cell Controller According to LUSTRE AST interpretation, Translation to B, and Code Generation to Java and C++ in Seconds with Number of Clock Steps (CK Steps), Memory Usage in KB, PI = Primitive Integer

Moving Item		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
(80 000 CK steps)	Runtime	31.69	6.35	2.52	1.25	1.20
	Memory	406 684	167 444	203 552	812	820
	CK Step/s	2524	12 598	31 746	64 000	66 666
Verify Moving Item		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
(80 000 CK steps)	Runtime	54.94	9.83	3.37	2.59	2.52
	Memory	585 596	167 492	253 544	840	848
	CK Step/s	1456	8138	23 738	30 888	31 746
Press		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
(51 200 CK steps)	Runtime	191.17	11.10	4.53	4.56	4.66
	Memory	303 176	167 872	402 496	1236	1248
	CK Step/s	267	4612	11 302	11 228	10 987
Rbase		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
(51 200 CK steps)	Runtime	116.66	8.60	3.91	3.41	3.48
	Memory	883 656	167 852	329 268	1228	1240
	CK Step/s	438	5953	13 094	15 014	14 712
Rgrips		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
(51 200 CK steps)	Runtime	76.38	5.43	3.17	2.06	2.08
	Memory	1 377 296	167 892	248 468	3016	3032
	CK Step/s	670	9429	16 151	24 854	24 615

As the production cell controller's LUSTRE nodes are implemented containing many logical operations, the speedup of the LUSTRE AST interpreter compared to the translation to B is relatively low. According to Moving Item and Verify Moving Item, the speedup is around five, while for Press, Rbase, and Rgrips, it is more than ten.

Execution in C++ leads to a better runtime than in Java for all production cell controller's nodes except Press. For Press, it seems that the execution in C++ is approximately as fast as in Java.

Larger programs also seem to be executed using the LUSTRE AST interpreter significantly faster than the corresponding translations to B. Submode Logic and Carlights are executed in C++ faster than in Java. For Pilot Flying, the execution in Java and C++ are approximately as fast. Docking Approach is even too large to be executed in Java, while its execution yields a fast runtime in C++. In contrast to Java, the LUSTRE AST interpreter does not have problems when simulating Docking Approach. Compared to Java, it seems that C ++ can handle large LUSTRE nodes that are not executed in a long-running loop.

Table 18: Runtimes of Larger LUSTRE Programs According to LUSTRE AST interpretation, Translation to B, and Code Generation to Java and C++ in Seconds with Number of Clock Steps (CK Steps), Memory Usage in KB

Carlights		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
(33 000 CK steps)	Runtime	157.75	6.38	3.83	2.54	2.58
	Memory	875 788	167 452	300 100	880	892
	CK Step/s	209	5172	8616	12 992	12 790
Pilot Flying		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
(8701 CK steps)	Runtime	434.41	10.99	5.97	5.99	6.05
	Memory	879 624	167 992	373 004	1108	1124
	CK Step/s	20	791	1457	1452	1251
Submode Logic		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
(8000 CK steps)	Runtime	52.06	3.38	2.76	1.38	1.38
	Memory	296 516	167 668	202 416	876	888
	CK Step/s	153	2366	2898	5797	5797
Docking Approach		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
(1099 CK steps)	Runtime	233.65	6.24	Code too large	5.67	5.56
	Memory	4 159 056	171 576	Code too large	2876	2888
	CK Step/s	4	176	Code too large	193	197

Considering the benchmarks results for programs with real numbers, one could assume that they are slower due to the real numbers. However, one must also note here that these programs are medium-sized to very large. As already analyzed in the microbenchmarks, real numbers do not have a great impact on performance.

Table 19: Runtimes of Interpreted LUSTRE Programs using Real Numbers with Number of Clock Steps (CK Steps), Memory Usage in KB, PI = Primitive Integer

Heat Controller		Lustre
(66 000 CK steps)	Runtime	14.97
	Memory	167 496
	CK Step/s	4408
Vehicle Speed		Lustre
(10 000 CK steps)	Runtime	26.92
	Memory	167 744
	CK Step/s	371
Cruise Controller		Lustre
(12 000 CK steps)	Runtime	26.50
	Memory	168 184
	CK Step/s	452

8.4 Model Checking

The performance of verification in PROB is analyzed in the context of explicit-state model checking and bounded model checking (BMC).
Invariant violations and state errors are checked by both explicit-state model checking and BMC. Additionally, explicit-state model checking also checks for deadlocks. All errors that are found in this section are either `nil` values in assertions, clocks, and output parameters of safety property violations. Deadlocks are possible in LUSTRE if there are no suceeding state statisfying the program's assumptions. The performance of BMC is investigated for the B translations only as it is not supported for the LUSTRE AST interpreter. In contrast, the performance of explicit-state model checking is analyzed for both approaches using PROB.
BMC is applied with a given search depth $k \in \{5, 20, 50\}$. This means that invariants and state errors are checked for the first k clock steps. Although this is a disadvantage of BMC, it can be used to check B machines with operations containing a high number of parameters. For state spaces that are known to be bounded to a search depth, this is the way how the program can be verified completely.
Each benchmark is applied five times and afterwards, the median of the walltimes is taken as the result. The timeout is set to 20 minutes for each benchmark despite Docking Approach GOAL (see Appendix G). There, the timeout is set to one hour.
To receive reproducible results, explicit-state model checking with both breadth-first search (BF) and depth-first search (DF) are applied to programs with errors. Programs without errors are model checked with `Mixed BF/DF`. Remark: For LUSTRE programs where the entire state space is checked, the corresponding B machines contain one more transition compared to model checking with the LUSTRE AST interpreter. This is the state after applying `SETUP_CONSTANTS`.
As explained in Section 6 and Section 7, verification of LUSTRE programs consists of checking safety properties and checking output parameters, clocks, and assertions for `nil` values. Nil checking of output parameters is relevant for main node only, while safety properties are only verified for verification nodes.
Lift, UMS, Train Speed, and SLOW TIME STABLE are programs with a low number of parameters. As shown in Table 20, Train Speed has a larger state space compared to the other programs. While the first three programs are implemented only using boolean parameters, the latter is implemented with an integer parameter. Thus, the number of possible parameter combinations for SLOW TIME STABLE is infinite. That is why explicit-state model checking is applied with integer parameters being in the interval from 0 to 10. As none of the four programs is a verification node, they are checked for `nil` values only. Checking for `nil` values in the output parameters is relevant for all four programs. In contrast, checking clocks and assertions for `nil` is relevant for SLOW TIME STABLE only.
The respective verification nodes for Train Speed and UMS are also taken into account. As there are assertions and safety properties that are added to the verification nodes, the sizes of their state space might differ from the nodes they verify. UMS Verification and Train Speed Verification are implemented without safety property violations. There is also a verification node for Lift analyzed which contains an error where the lift is applied to a building with 48 floors. Despite, there are also versions of Train Speed Verification and UMS Verification considered that are implemented containing safety property vio-

lations. Furthermore, there are also modified versions of UMS Verification and SLOW TIME STABLE containing `nil` values in assertions and clocks respectively.
Press, Rbase, and Rgrips consist of a high number of parameters and thus also a high number of transitions. These three nodes and Moving Item are checked for `nil` values of output parameters. Compared to Press, Rbase, and Rgrips, Moving Item and Verify Moving Item have fewer states and transitions. The state space of Verify Moving Item is smaller than the one from Moving Item as it contains additional safety properties.
Carlights consists of two assertions and three local clocks which must all be checked for `nil`. It does not contain any safety properties to be verified. For Pilot Flying, Submode Logic, and Docking Approach, there are versions investigated which are implemented with and without safety properties. There are also modified versions containing violated safety properties. Note that these are only properties that are not always held by the system. Furthermore, it is also analyzed whether safety property violations can be found easier when they are decomposed. Table 24 shows the result of explicit-state model checking and BMC for Pilot Flying, Submode Logic, and Docking Approach as well as the verification nodes for the first two programs with composed safety properties. The corresponding nodes with decomposed safety properties are shown in Appendix G with some of them containing safety property violations. There, also a modified version with fewer parameters named Docking Approach GOAL is shown from which it can find the trace for simulation. Despite `StageTransition`, all integer parameters in the original version are set to 2 and defined as local variables. Furthermore, the safety property is defined as `not(JointMission)`. This makes it possible to find a trace where the Docking Approach is completed. As mentioned before, its timeout is set to one hour for demonstration that such a path can be found with model checking. Docking Approach contains a high number of parameters with some of them being integers. Similar to SLOW TIME STABLE, the integer parameters are limited to an interval (0 to 11 for `StageTransition`, 0 to 2 for other integer parameters) for explicit-state model checking. Although Submode Logic is a larger LUSTRE program, it has a small state space. In contrast, the state spaces of Pilot Flying and Docking Approach are very large, possibly infinite.
As Heat Controller, Vehicle Speed, and Cruise Controller are implemented with real numbers, they are verified with explicit-state model checking using the LUSTRE AST interpreter only. Integer and real values as parameters are limited to an input set which is chosen as follows: Heat Controller expects input between 14.0 to 26.0 in 0.5 steps for real numbers. For Cruise Controller, real values are limited between 0.0 and 100.0 in steps of 5.0, while integer values are limited from 0 to 8. So, explicit-state model checking is applied to a part of the state space only. Although the programs' inputs are limited, it still results in a large (or even infinite) state space.
Explicit-state model checking using the LUSTRE AST interpreter yields a significant faster runtime than the translations to B. The speedup is lower for programs that are implemented using many boolean values.
It looks like explicit-state model checking with both approaches can be applied well to verify programs with finite state spaces. As one can see, all of the smaller programs can be verified under a second using the LUSTRE AST interpreter. According to the B translations, Train Speed and its verification node require nearly ten seconds.
BMC seems to be less suitable for verification of programs with finite state space. On the one hand, the programs are only verified for a certain number of steps. On the other

hand, BMC seems to have some problems for smaller programs with safety properties such as UMS Verification and Train Verification. This might be due to the reason that BMC cannot handle the values' representation in B so well. There is also a problem that no path with more than one step can be generated for SLOW TIME STABLE.
Errors in the modified versions can be detected and counterexamples can be generated with explicit-state model checking fast. BMC is also able to detect errors that are located in higher levels of the state space fast. However, the modified version of Train Speed Verification contains an error in depth 11 where BMC requires significantly more time to detect the error.

Table 20: Runtimes of Explicit-State Model Checking and BMC for Smaller Programs in Seconds with Number of States and Transitions, Memory Usage in KB, TO = Timeout

Lift		B	Lustre	BMC(5)	BMC(20)	BMC(50)
(401 states, 1605 transitions)	Runtime	0.68	0.08	0.31	1.08	5.03
	Memory	170 584	167 200	168 760	168 760	168 804
	States/s	589	5012	-	-	-
	Transitions/s	2360	20 062	-	-	-
SLOW TIME STABLE [12]		B	Lustre	BMC(5)	BMC(20)	BMC(50)
(24 states, 1057 transitions)	Runtime	0.29	0.06	0.29 after Level 1	0.29 after Level 1	0.29 after Level 1
	Memory	170 760	167 656	169 108	169 108	169 108
	States/s	82	787	-	-	-
	Transitions/s	3644	34 663	-	-	-
UMS		B	Lustre	BMC(5)	BMC(20)	BMC(50)
(6 states, 193 transitions)	Runtime	0.04	0.02	0.35	1.69	8.61
	Memory	169 112	167 188	168 684	168 688	168 696
	States/s	150	400	-	-	-
	Transitions/s	4825	17 616	-	-	-
Train Speed		B	Lustre	BMC(5)	BMC(20)	BMC(50)
(2020 states, 8081 transitions)	Runtime	8.13	0.47	0.53	3.08	19.58
	Memory	175 708	169 592	168 960	168 956	168 976
	States/s	248	4297	-	-	-
	Transitions/s	993	17 193	-	-	-
UMS Verification		B	Lustre	BMC(5)	BMC(20)	BMC(50)
(111 states, 646 transitions)	Runtime	2.44	0.91	11.15	TO at Level 10	TO at Level 10
	Memory	172 000	167 192	170 680	170 684	170 684
	States/s	45	121	-	-	-
	Transitions/s	264	709	-	-	-
Train Speed Verification		B	Lustre	BMC(5)	BMC(20)	BMC(50)
(2020 states, 8081 transitions)	Runtime	9.27	0.62	0.74	TO at Level 12	TO at Level 12
	Memory	176 032	170 116	169 044	169 152	169 152
	States/s	217	3258	-	-	-
	Transitions/s	871	13 033	-	-	-

[12] BMC cannot generate a path with a length ≥ 2

Table 21: Runtimes of Explicit-State Model Checking and BMC for Smaller Programs with Errors in Seconds with Number of States and Transitions, Memory Usage in KB, TO = Timeout, EF = Error Found, BF = Breadth-First Serach, DF = Depth-First Search

Lift Verification Error		B BF	B DF	Lustre BF	Lustre DF	BMC(5)	BMC(20)	BMC(50)
	Runtime	0.48 (EF)	0.19 (EF)	0.07 (EF)	0.07 (EF)	0.42	TO at Level 15	TO at Level 15
	Memory	170 496	169 996	167 736	167 256	168 984	169 476	169 476
	States	146	50	144	146	-	-	-
	Transitions	441	153	434	440	-	-	-
	States/s	304	263	2057	2085	-	-	-
	Transitions/s	918	805	6200	6285	-	-	-
SLOW TIME STABLE Clock Nil[13]		B BF	B DF	Lustre BF	Lustre DF	BMC(5)	BMC(20)	BMC(50)
	Runtime	0.08 (EF)	0.07 (EF)	0.04 (EF)	0.04 (EF)	0.48 at Level 1 (EF)	0.48 at Level 1 (EF)	0.48 at Level 1 (EF)
	Memory	169 636	169 624	167 160	167 220	169 044	169 044	169 044
	States	0	0	1	1	-	-	-
	Transitions	2	2	45	45	-	-	-
	States/s	0	0	25	25	-	-	-
	Transitions/s	25	28	1125	1125	-	-	-
UMS Verification Error		B BF	B DF	Lustre BF	Lustre DF	BMC(5)	BMC(20)	BMC(50)
	Runtime	0.17 (EF)	0.13 (EF)	0.07 (EF)	0.05 (EF)	0.75 after Level 2 (EF)	0.75 after Level 2 (EF)	0.75 after Level 2 (EF)
	Memory	170 008	169 964	167 252	167 224	169 644	169 644	169 644
	States	6	3	7	3	-	-	-
	Transitions	226	130	257	129	-	-	-
	States/s	35	23	100	60	-	-	-
	Transitions/s	1329	1000	3671	2580	-	-	-
UMS Verification Assertion Nil		B BF	B DF	Lustre BF	Lustre DF	BMC(5)	BMC(20)	BMC(50)
	Runtime	0.03 (EF)	0.03 (EF)	0.05 (EF)	0.05 (EF)	TO at Level 1 (EF)	TO at Level 1 (EF)	TO at Level 1 (EF)
	Memory	170 908	170 920	167 256	167 244	170 392	170 392	170 392
	States	0	0	1	1	-	-	-
	Transitions	2	2	33	33	-	-	-
	States/s	0	0	20	20	-	-	-
	Transitions/s	66	66	660	660	-	-	-
Train Speed Verification Error		B BF	B DF	Lustre BF	Lustre DF	BMC(5)	BMC(20)	BMC(50)
	Runtime	0.15 (EF)	4.47 (EF)	0.04 (EF)	0.04 (EF)	0.74	462.18 after Level 11 (EF)	462.18 after Level 11 (EF)
	Memory	169 492	172 608	167 244	167 232	169 100	169 424	169 424
	States	22	1011	23	11	-	-	-
	Transitions	94	4050	97	49	-	-	-
	States/s	146	226	575	275	-	-	-
	Transitions/s	626	906	2425	1225	-	-	-

As Press, Rbase, and Rgrips are implemented with a high number of parameters, the resulting state space consists of many transitions. Here, one can see that explicit-state model checking can evaluate a high number of transitions per second, while the number of evaluated states per second is quite low. Unlike UMS Verification, BMC seems to handle the production cell controller's nodes well. Both programs are implemented with boolean values only. It seems that UMS Verification has much more branches to be covered when generating test cases in the algorithm of BMC. So, it cannot be said whether BMC can be applied to pure boolean nodes well.

Furthermore, there is a version of Moving Item taken into account which leads to a `nil` value in the output parameters. This can be detected by explicit-state model checking as well as BMC. For the modified version of Rbase, it cannot be outlined whether depth-first search or breadth-first search perform better. While breadth-first detects the error faster using the LUSTRE AST interpreter, it is the other way around for the B translation.

[13] BMC cannot generate a path with a length ≥ 1

Table 22: Runtimes of Explicit-State Model Checking and BMC for Production Cell Controller in Seconds with Number of States and Transitions, Memory Usage in KB, TO = Timeout

Moving Item		B	Lustre	BMC(5)	BMC(20)	BMC(50)
(33 states, 265 transitions)	Runtime	0.13	0.04	0.71	9.05	64.04
	Memory	169 424	167 200	169 016	169 028	169 080
	States/s	253	825	-	-	-
	Transitions/s	2038	6625	-	-	-
Verify Moving Item		B	Lustre	BMC(5)	BMC(20)	BMC(50)
(52 states, 417 transitions)	Runtime	0.29	0.07	1.34	25.35	206.00
	Memory	169 716	167 192	169 280	169 300	169 384
	States/s	179	742	-	-	-
	Transitions/s	1437	5957	-	-	-
Press		B	Lustre	BMC(5)	BMC(20)	BMC(50)
(653 states, 334 337 transitions)	Runtime	219.71	59.75	2.53	66.76	511.18
	Memory	487 088	359 856	169 572	169 712	170 312
	States/s	2	10	-	-	-
	Transitions/s	1521	5595	-	-	-
Rbase		B	Lustre	BMC(5)	BMC(20)	BMC(50)
(455 states, 232 961 transitions)	Runtime	97.33	32.25	1.81	39.97	322.73
	Memory	393 064	303 004	169 416	169 524	169 776
	States/s	4	14	-	-	-
	Transitions/s	2393	7223	-	-	-
Rgrips		B	Lustre	BMC(5)	BMC(20)	BMC(50)
(89 states, 182 273 transitions)	Runtime	74.05	26.62	0.66	8.29	67.12
	Memory	366 888	283 980	169 072	169 124	169 280
	States/s	1	3	-	-	-
	Transitions/s	2461	6847	-	-	-

Table 23: Runtimes of Explicit-State Model Checking and BMC for Production Cell Controller with Errors in Seconds with Number of States, Memory Usage in KB, TO = Timeout, EF = Error Found, BF = Breadth-First Serach, DF = Depth-First Search

Moving Item Nil		B BF	B DF	Lustre BF	Lustre DF	BMC(5)	BMC(20)	BMC(50)
	Runtime	0.05 (EF)	0.04 (EF)	0.03 (EF)	0.05 (EF)	0.30 at Level 1 (EF)	0.30 at Level 1 (EF)	0.30 at Level 1 (EF)
	Memory	169 396	169 396	167 228	167 244	169 308	169 308	169 308
	States	5	1	1	31	-	-	-
	Transitions	50	18	17	257	-	-	-
	States/s	100	25	33	620	-	-	-
	Transitions/s	1000	450	566	5140	-	-	-
Rbase Nil		B BF	B DF	Lustre BF	Lustre DF	BMC(5)	BMC(20)	BMC(50)
	Runtime	3.93 (EF)	0.45 (EF)	0.17 (EF)	30.35 (EF)	0.38 at Level 1 (EF)	0.38 at Level 1 (EF)	0.38 at Level 1 (EF)
	Memory	178 796	171 012	167 248	300 964	169 632	169 632	169 632
	States	17	1	1	446	-	-	-
	Transitions	9218	1026	1025	228 865	-	-	-
	States/s	4	2	5	14	-	-	-
	Transitions/s	2345	2280	6029	7540	-	-	-

Explicit-state model checking is also able to verify larger programs with finite state spaces such as Carlights, Submode Logic, and Submode Logic Verification. Although BMC can check Submode Logic for `nil` values, the safety property of Submode Logic Verification can only be verified for the first eight steps. Furthermore, BMC is not able to check the clocks in Carlights for `nil`. The process is killed with approximately 5 GB of memory usage without verifying any step. BMC possibly has problems dealing with `IF` substitutions similar to SLOW TIME STABLE.
According to Pilot Flying, explicit-state model checking can only verify a part of the program as it has a very large (possibly infinite) state space. BMC is also only applicable to Pilot Flying and its verification node in the first four and seven steps respectively. Again, Docking Approach can be checked for `nil` values of output parameters in the first 14 clock steps. As it is implemented containing a high number of parameters, the benchmark leads to a timeout when starting explicit-state model checking for both approaches in PROB.

Table 24: Runtimes of Explicit-State Model Checking and BMC for Larger Programs in Seconds with Number of States and Transitions, Memory Usage in KB, TO = Timeout

Carlights [14]		B	Lustre	BMC(5)	BMC(20)	BMC(50)
(1611 states, 53 164 transitions)	Runtime	57.61	20.94	TO at Level 1	TO at Level 1	TO at Level 1
	Memory	199 252	186 972	4 816 076	4 816 076	4 816 076
	States/s	27	76	-	-	-
	Transitions/s	922	2538	-	-	-
Pilot Flying [15]		B	Lustre	BMC(5)	BMC(20)	BMC(50)
	Runtime	TO after 20 min	TO after 20 min	4.45	TO at Level 7	TO at Level 7
	Memory	515 548	596 188	170 580	170 840	170 840
	States	16 476	43 550	-	-	-
	Transitions	208 750	455 071	-	-	-
	States/s	13	36	-	-	-
	Transitions/s	173	379	-	-	-
Pilot Flying Verification [16]		B	Lustre	BMC(5)	BMC(20)	BMC(50)
	Runtime	TO after 20 min	TO after 20 min	TO at Level 4	TO at Level 4	TO at Level 4
	Memory	526 044	647 804	170 916	170 916	170 916
	States	12 651	35 200	-	-	-
	Transitions	164 304	378 603	-	-	-
	States/s	10	29	-	-	-
	Transitions/s	136	315	-	-	-
Submode Logic		B	Lustre	BMC(5)	BMC(20)	BMC(50)
(7 states, 57 transitions)	Runtime	0.12	0.04	0.72	8.38	63.13
	Memory	170 132	169 776	169 424	169 444	296 556
	States/s	58	175	-	-	-
	Transitions/s	475	1425	-	-	-
Submode Logic Verification		B	Lustre	BMC(5)	BMC(20)	BMC(50)
(7 states, 57 transitions)	Runtime	0.12	0.04	1.86	TO at Level 8	TO at Level 8
	Memory	169 724	167 388	169 848	169 820	169 820
	States/s	58	175	-	-	-
	Transitions/s	475	1425	-	-	-
Docking Approach		B	Lustre	BMC(5)	BMC(20)	BMC(50)
	Runtime	TO at Start	TO at Start	101.46	TO at Level 14	TO at Level 14
	Memory	178 272	344 244	303 580	592 872	592 872

Safety properties are also decomposed for the verification nodes of Pilot Flying, Submode Logic, and Docking Approach with the hope that BMC can handle it better (see Appendix G). Here, some properties which are not always true are also taken into account. For Pilot Flying and Submode Logic, it turns out that explicit-state model checking of both approaches and BMC can find these errors in a reasonable time if they are not located deep in the state space. There is also a property considered in Pilot Flying where only breadth-first search using the LUSTRE AST interpreter can find a counterexample with a path with five clock step operations within 20 minutes. For demonstration, also the LUSTRE program Docking Approach GOAL is taken into account. Here, the safety property is set to `not(JointMission)` which means that there exists no path to finish the Docking Approach. The LUSTRE AST interpreter can find a path finishing the Docking Approach within 27 clock steps (28 operations including initialization) using breadth-first search. Again, with depth-first search, it was able to find a path applying 1099 clock steps. This path was also chosen as the simulation benchmark. Note that it is necessary to set the timeout to one hour to find both paths. Decomposing the safety properties turns out not to be so helpful for BMC. However, BMC is not able to verify the most properties for more than eight steps. Therefore, no counterexamples with a path longer than the reached search depth can be found within 20 minutes.

As shown in Table 25, LUSTRE programs containing real numbers cannot be verified using explicit-state model checking well. Although the input for real numbers and integers

[14]Process of BMC killed at Level 1

[15]Unknown number of states and transitions, possibly infinite

[16]Unknown number of states and transitions, possibly infinite

are limited, explicit-state model checking of these programs leads to the state space explosion problem. One can also see that the memory usage is very high as many transitions are evaluated. According to the future, it could be an approach to abstract real numbers before model checking. This could be one way to address the state space explosion problem as described in [34].

Table 25: Runtimes of Explicit-State Model Checking and BMC for Programs using Real Numbers in Seconds with Number of States and Transitions, Memory Usage in KB, TO = Timeout

Heat Controller		Lustre
	Runtime	TO after 20 min
	Memory	1 328 464
	States	225
	Transitions	3 890 626
	States/s	< 1
	Transitions/s	3242
Vehicle Speed		Lustre
	Runtime	TO after 20 min
	Memory	4 940 128
	States	122 250
	Transitions	365 903
	States/s	101
	Transitions/s	304
Cruise Controller		Lustre
	Runtime	TO after 20 min
	Memory	502 824
	States	50
	Transitions	592 705
	States/s	< 1
	Transitions/s	493

9 Related Work

During this work, a LUSTRE interpreter for PROB and a translator from LUSTRE to B named LUSTRE2B were implemented. This enables simulation and verification of LUSTRE programs. Simulation includes the usage of the PROB animator and the code generator B2PROGRAM, while verification includes different model checking techniques.
This work is compared to other works on LUSTRE simulators, code generators, and verification tools, especially model checkers in this section.

9.1 Comparison with Other Lustre Simulators and Code Generators

Simulators and code generators can be used for simulation of LUSTRE programs. There are also LUSTRE code generators that target code for embedded systems. On the one hand, there are code generators that generate code from LUSTRE to other programming languages. On the other hand, there are also code generators that target LUSTRE e.g. translators from Simulink to LUSTRE.

LUCIOLE is one of the simulators that is introduced with the LUSTRE toolbox. It provides a graphical user interface with a panel where the user adjusts the parameter values for the next clock step. After applying the clock step, one can see the output values. Furthermore, LUCIOLE also enables real-time simulation where a clock step is automatically applied after passing a specific time. In combination with SIM2CHRO, one can view a variable's values that are plotted against the time in a cartesian coordinate system. [28]

As achieved in this work, PROB Tcl/Tk and PROB2 JavaFX UI can be used to simulate LUSTRE programs. This can either be done as translated B models or with the LUSTRE AST interpreter. During the simulation, the user can trigger clock step operations, or view the current state, and the history of clock steps that leads to the current state. While LUCIOLE shows the main node's output values only, PROB provides the possibility to view the state of all nodes in the program. There, each node can be seen as an encapsulated component. One can also see the variables' values that make up the memory of each node. Furthermore, clocks, and variables that make up the safety properties are always part of the program state. Both GUIs also support many dot-based graphical visualizations e.g. visualization of the state space. PROB2 JavaFX UI also provides the opportunity to view the variables' values in a cartesian coordinate system. This feature is supported for the translated B models only.

Another way to simulate LUSTRE programs is provided by the LUX simulator which is also part of the LUSTRE toolbox. LUX expects a LUSTRE file in the OC format and generates C code. The OC format is an intermediate representation for LUSTRE programs that is used by many tools in the LUSTRE toolbox [35]. With the generated C code, one can simulate the LUSTRE program each clock step interactively. [28]
As mentioned before, this is also supported by PROB CLI, PROB Tcl/Tk, and PROB2 JavaFX UI. This work also aims translation of LUSTRE programs to a subset of B which also makes code generation to Java and C++ using B2PROGRAM feasible. The generated Java and C++ programs can then be used for simulation. Note that the generated code from B2PROGRAM is the raw program only. Implementing the *main function* is up to the user. So, one has the opportunity either to execute a sequence of clock step operations and then print the values of the variables, or to implement some kind of interactive simulation.

There also exists a formally verified compiler for LUSTRE that is built on COMPCERT. It aims C code generation for embedded systems. Both the generated C code as well as the compiler are verified. [36]
Generally, most code generators generate code from a verified specification but are not verified. Especially for embedded systems, the verification of memory usage is of significant importance. Once the system runs out of memory, it crashes and might lead to an accident. Another code generator for embedded systems is presented in [37]. It is part of the ZELUS compiler and generates code from models written in SCADE 6. Apart from code generation for embedded systems, it also targets code generation for simulation platforms.
In contrast, B2PROGRAM generates code which uses high-level libraries that are not verified for the B types. Thus, the generated code is not allowed to be used for embedded systems. Instead, B2PROGRAM aims code generation for simulation, monitoring, and data validation [7].

Furthermore, there are code generators that translate Simulink and Stateflow models to LUSTRE programs [38, 39, 40, 41]. Embedded Systems are often used in the context of engineering and physics. So, the idea is modeling in a mathematical language and then translate to a programming language that is used in the context of embedded systems. This also makes it possible to use simulators, code generators, and model checkers that are implemented for LUSTRE. E.g. COCOSIM [41] translates Simulink models to LUSTRE

for verification using one of the following model checkers: ZUSTRE, KIND2, and JKIND. Furthermore, COCOSIM also aims code generation from Simulink to C and Rust via LUSTRE for simulation.
In contrast, this work brings LUSTRE to a platform that uses *set theory* and *first-order logic* which are the mathematical ideas the B-Method and PROB are built at. Note that these mathematical ideas differ from those for engineering and physics. The translation of LUSTRE to B in this work also makes code generation to Java and C++ using B2PROGRAM feasible.

Some works aim generation of sequential programs for LUSTRE [11]. This results in single-loop code and automaton-like code. In contrast, this work rewrites the LUSTRE AST such that it can be interpreted sequentially. Furthermore, sequential B code is generated from the resulting AST using LUSTRE2B. A good performance can then be achieved by generating Java and C++ code using B2PROGRAM.

9.2 Comparison with Lustre Model Checkers

There are several model checkers for LUSTRE including explicit-state model checkers and symbolic model checkers.

One of the explicit model checkers is LESAR which is presented by Pascal Raymond [34]. To deal with the state space explosion problem, LESAR abstracts constructs using integers and real numbers. This results in LUSTRE programs containing boolean values only. Furthermore, it is possible to abstract infinite state spaces to finite state spaces. Afterwards, explicit-state model checking is applied to an over-approximated transition system.
According to the LUSTRE AST interpreter, explicit-state model checking and LTL model checking are supported. Additionally, symbolic model checking can also be applied to the translated B models.
This work reduces the state space by detecting which variables are necessary to be stored in the program's state. So, the focus is set on variables that are relevant for the output parameters, safety properties, clocks, and the nodes' memory. As result, explicit-state model checking can find some errors for programs with fewer parameters and larger state spaces in a reasonable time. Nevertheless, this work still struggles with larger state spaces e.g. BMC is only able to verify some examples for the first clock steps.

One of the symbolic model checkers is PKIND which is presented in [42]. It is built on the SMT solvers CVC3 and Yices and uses parallelization to improve the performance. PKIND applies k-Induction with invariant generation to verify safety properties of LUSTRE programs. Invariant generation and parallel execution of model checking make it possible to improve the performance of model checking with PKIND.
In contrast, this work also enables symbolic model checking techniques such as BMC, k-Induction, and IC3 when translating LUSTRE programs to B. While PKIND can verify properties of Docking Approach in a reasonable time, this work can only verify LUSTRE programs using BMC for the first clock steps. Furthermore, PKIND can find counterexamples for false properties in Docking Approach that are located deeper in the state space. Errors in Docking Approach that are reached after seven or more steps cannot be detected by BMC in this work.
The successor of PKIND is KIND 2 which also applies invariant generation and parallel

model checking. It also supports BMC and IC3 in addition to k-Induction. Apart from CVC3 and Yices, KIND 2 also supports the SMT solver Z3. Like PKIND, KIND 2 is also able to verify the safety properties of LUSTRE programs with infinite state spaces. It is also applied to properties in Docking Approach which even leads to a better result than PKIND.
Another infinite-state model checker for LUSTRE is JKIND which is presented in [43]. It is also based on SMT solvers and is built with parallel engines to improve the performance. It also supports symbolic model checking techniques such as k-Induction and BMC. Similarly, JKIND applies invariant generation to improve the performance of model checking. As described in the paper, JKIND can keep up with the other symbolic model checkers according to the performance. A special feature of this model checker is that it can be compiled and integrated into Java programs.

9.3 Comparison with Other Verification Tools for Lustre

Other approaches for verification of LUSTRE programs are abstract interpretation and proving. Abstract interpretation is a technique that over-approximates a program for verification. A tool for abstract interpretation of LUSTRE programs is NBAC that is developed by Bertrand Jeannet [44]. Another work that makes use of abstract interpretation and SMT solving is presented in [45]. There also exists a PVS proof obligation generator for LUSTRE which targets the proving tool GLOUPS [46]. Again, the approach described in [47] verifies LUSTRE proof certificates with SMT solving. In comparison to this work, it does not apply abstract interpretation or proving. Nevertheless, there is still the opportunity to prove the translated B models in provers such as ATELIERB.

10 Conclusion and Future Work

The purpose of this thesis was extending PROB by LUSTRE for simulation and verification. During this work, two approaches have been outlined to achieve this: implementing a PROB interpreter for LUSTRE and translating LUSTRE programs to B. As a result, a LUSTRE AST interpreter for PROB and a translator from LUSTRE to B named LUSTRE2B are implemented successfully.
Furthermore, both approaches share the same LUSTRE parser which was also implemented in this work. Some optimizations are implemented in the parser such that interpretation and translation to B are simplified. On the one hand, the order of assertions and equations in the AST is rewritten such that it can be interpreted sequentially, and such that sequential B code can be generated. On the other hand, operands of temporal expressions are rewritten following the *substitution principle* such that they are always stored in variables. This makes it possible to access previous values of flows that are not stored in variables which might be required for the calculation of the following cycles. Furthermore, the parser also preprocesses data for the state representation of LUSTRE programs in the PROB animator. Output variables, clocks, variables in `pre`, and variables that are sampled on a slower clock than the *basic clock* are always stored in the program's state. Additionally, the final state representation in the PROB animator also contains the invoked nodes' states, and a flag for each clock describing whether its next

cycle is the first cycle.
For the translation to B, a B library is implemented containing B functions imitating LUSTRE's semantics. Following this approach, the translation of B expressions can be simplified by invoking the implemented functions. The main difference is that LUSTRE expressions can accept `nil` values which are modeled as $\varnothing$. Values other than `nil` are modeled as a relation $\{ref \to val\}$ in B.
It is also achieved to translate LUSTRE programs to a subset of B that makes the application of B2PROGRAM feasible. As a result, this work also enables code generation of LUSTRE via B to Java and C++ for simulation. According to the simulation in the PROB animator, using the LUSTRE AST interpreter leads to a significantly better performance than the translated B models. In contrast, better performance can be achieved by translating the LUSTRE programs to B for code generation to Java and C++. Furthermore, it seems that the translations to C++ yield better performance than Java for larger programs. Again, execution in Java possibly leads to a slightly better performance than C++ for smaller programs where many *clock step operations* are applied.
This work also makes it possible to verify LUSTRE programs using different model checking techniques. Explicit-state model checking and LTL model checking are supported for both approaches. Additionally, symbolic model checking is also supported by translating the LUSTRE programs to B. Similar to simulation, the LUSTRE AST interpreter yields a better runtime than the translations to B according to explicit-state model checking. Errors in smaller programs with finite state spaces can be detected using explicit-state model checking by both approaches well. In contrast, errors in larger (or infinite) state spaces can only be detected in a reasonable time if they are not located deep in the state space. Nevertheless, there are still problems when verifying LUSTRE programs with larger or even infinite state spaces. In this work, it seems that BMC can only detect errors that are located in the first few clock steps of the program.
Currently, only a subset of LUSTRE is supported. Other features such as functions, constants, arrays, records, user-defined types, parametric nodes, recursive nodes could be implemented in the future. Also, it is not possible to include many LUSTRE files or define packages which could also be implemented as future work.
Node expansions are not applied to the AST yet. This could be a possibility to extend the supported subset of LUSTRE. For now, LUSTRE programs are translated to B using sequential substitutions. According to the future, it would also be possible to apply the *substitution principle* on the AST and then to generate parallel substitutions instead. Here, one could also analyze the performance. Furthermore, this idea would also provide the opportunity to translate LUSTRE to TLA. The translated TLA modules can then be verified using the TLC model checker. Following this approach would also require to expand node calls.
In general, the performance of both model checking and simulating LUSTRE programs could be improved in the future. One could also take the implementation of a static analyzer for `nil` values into account to reduce the verifications that are applied at runtime.
Furthermore, this work also enables simulation and verification of LUSTRE programs in both PROB GUIs (PROB Tcl/Tk and PROB2-UI). As LUSTRE is often linked with graphical visualizations, this could be implemented in the future. It would also be possible to extend LUSTRE by SCADE which is a graphical modeling language that bases on LUSTRE.

A Token Representations and Regex

A.1 Keyword and Operator Tokens

Token	Representation	Regex
node	node(Pos)	"node"
var	var(Pos)	"var"
let	let(Pos)	"let"
tel	tel(Pos)	"tel"
returns	returns(Pos)	"returns"
int	integer(Pos)	"int"
bool	boolean(Pos)	"bool"
real	real(Pos)	"real"
assert	assert(Pos)	"assert"
if	if(Pos)	"if"
then	then(Pos)	"then"
else	else(Pos)	"else"
+	plus(Pos)	"+"
-	minus(Pos)	"-"
*	mul(Pos)	"*"
/	div(Pos)	"/"
%, mod	mod(Pos)	("%" | "mod")
<	less(Pos)	"<"
<=	lessequal(Pos)	"<="
>	greater(Pos)	">"
>=	greaterequal(Pos)	">="
=	equal(Pos)	"="
<>	unequal(Pos)	"<>"
and	and(Pos)	"and"
or	or(Pos)	"or"
not	not(Pos)	"not"
pre	pre(Pos)	"pre"
->	fby(Pos)	"->"
when	when(Pos)	"when"
current	current(Pos)	"current"
ˆ	caret(Pos)	"^"
..	dotdot(Pos)	".."

A.2 Identifier and Literal Tokens

Token	Representation	Regex
Identifier	identifier(Pos, ID)	[_a-zA-Z][_a-zA-Z0-9]*
Integer Literal	integer_val(Pos, Val)	[0-9]$^+$
Real Literal	real_val(Pos, Val)	[0-9]$^+$ "." [0-9]$^+$
true	true(Pos)	"true"
false	false(Pos)	"false"
nil	nil(Pos)	"nil"

Remark: Condition for identifier is that it is not any of keywords, operators or literals

A.3 Separator Tokens

Token	Representation	Regex
(	lpar(Pos)	"("
)	rpar(Pos)	")"
[	lbra(Pos)	"["
]	rbra(Pos)	"]"
,	comma(Pos)	","
:	colon(Pos)	":"
;	semicolon(Pos)	";"

A.4 Ignored Tokens

Token	Representation	Regex
Whitespace	-	(" " | "\t" | "\n")$^+$
Single-Line Comment	-	"-""-"[^"\n"]*"\n"
Multi-Line Comment	-	("(*"(.)*"*)" | "/*"(.)*"*/")

B Not Supported Lustre Constructs

Not supported Lustre construct
Constants
Some operators (div, merge, =>, xor, #, nor, fby[17])
Explicit Type Casting Operators (int, real)
Static Recursion, with Operator, Recursive Nodes
Arrays, Array Iterators
User-defined types (structs, records, enums)
Extern Nodes
Functions
Genericity of Nodes and Functions
Packages and Models, Inclusion of Other Lustre Files
Pragmas

Table 26: Overview of Not Supported Lustre Constructs

Note that each clock must be used as a boolean variable in the supported subset of LUSTRE. Furthermore, there is a restriction on node calls where the equations might be causal (see Section 5.4.3).

[17]This operator combines -> and pre and is a more user-friendly version of ->

C Interpreter Semantics

C.1 Auxiliary Definitions and Rules

$$ast(node) := \text{AST node of a L}\textsc{ustre}\ node$$

$$clock(E) := cl(E) := \text{Clock of E}$$

$$\tau(E) := \text{Type of E}$$

$$literal(E) := \text{E is literal}$$

$$variable(E) := \text{E is variable}$$

$$basic(C) := \text{C is basic clock}$$

$$main\_node(node) := \text{node is main node}$$

$$main := \text{main node/chosen node}$$

$$verification\_node(node) := \text{node is verification node}$$

$$output(node(p_1, \dots, p_n)) := \text{Output Identifiers of Node Call } node(p_1, \dots, p_n)$$

$$\begin{gathered} parameters(\textbf{node}\ nodec\ (p_1; \dots; p_n)\ ...) := parameters(p_1; \dots; p_n) \\ parameters(p_1; \dots; p_n) := parameters(p_1), \dots, parameters(p_n) \\ parameters(v_1, \dots, v_n : \tau) := v_1, \dots, v_n \\ parameters(v_1, \dots, v_n : \tau\ \textbf{when}\ C) := v_1, \dots, v_n \end{gathered}$$

$$\begin{gathered} oparameters(\textbf{node}\ ...\ \textbf{returns}(p_1; \dots; p_n)\ ...) := oparameters(p_1; \dots; p_n) \\ oparameters(p_1; \dots; p_n) := oparameters(p_1), \dots, oparameters(p_n) \\ oparameters(v_1, \dots, v_n : \tau) := v_1, \dots, v_n \\ oparameters(v_1, \dots, v_n : \tau\ \textbf{when}\ C) := v_1, \dots, v_n \end{gathered}$$

$$id(node(p_1, \dots, p_n)) := \text{Instance ID of Node Call } node(p_1, \dots, p_n)$$

$$svar_{node} := \text{state variables of node}$$

$$non - svar_{node} := \text{non-state variables of node}$$

$$clocks_{node} := \text{clocks of node}$$

$$chierarchy_{node} := \text{clock hierarchy of node}$$

$$\sigma_{root} := \text{Lustre Root}$$

$$\sigma_{abort} := \text{Abort State}$$

$$\sigma_{cut-off} := \text{Cut-Off State}$$

$$\sigma_{empty} := \varnothing$$

$$\sigma_{state} := (\sigma_{name}, \sigma_{svar}, \sigma_{clocks}, \sigma_{nodes})$$

$$\sigma := (\sigma_{name}, \sigma_{svar}, \sigma_{non-svar}, \sigma_{clocks}, \sigma_{nodes}, \sigma_{his-var}, \sigma_{ctf})$$

σ_{name} is the node's name.

σ_{svar}, $\sigma_{non-svar}$, σ_{clocks}, σ_{nodes} are the respective environments for *state variables*, *non-state variables*, the clocks' initialization statuses, and invoked node instances. They are relations in the mathematical notation. According to the implementation, each environment is an AVL tree. AVL trees are represented as relations in the mathematical notation.

The empty environment σ_{empty} and the *root state* σ_{root} are always globally accesible.

Furthermore, each node's AST (via $ast(node)$) and the preprocessed data for each node ($svar_{node}$, $non-svar_{node}$, $clocks_{node}$, $chierarchy_{node}$) are globally accesible.

σ_{ctf} is a flag for cutting off unreachable states, $\sigma_{his-var}$ is the environment of *state variables* in the previous cycle.

σ_{ctf}, $\sigma_{his-var}$, and $\sigma_{non-svar}$ do not represent the program's state. They are only relevant for the calculation of the succeeding state.

$\rightsquigarrow_{S;}$ denotes modification of a state after interpreting S; (used in the form $\sigma \rightsquigarrow_{S;} \sigma'$)

$\rightarrow_{\alpha(p_1,\ldots,p_n)}$ denotes modification of a state after applying an action α with given parameters (e.g. updating values or simulating node calls)

$\Rightarrow$ denotes an evaluation without modifying the state (used in the form $\langle x, \sigma\rangle \Rightarrow E$, where x is a variable, σ a state and E the result)

$\Rightarrow_\kappa$ denotes an evaluation in a specific context κ (e.g. checking whether or clock is active or checking safety properties)

$\sigma \lhd\!\!\!+ \{v \mapsto E\}$ overrides value of v to E in σ.

$$node(\sigma_{name}, \sigma_{svar}, \sigma_{clocks}, \sigma_{nodes}) := \sigma_{name}$$

$$node(\sigma_{name}, \sigma_{svar}, \sigma_{non-svar}, \sigma_{clocks}, \sigma_{nodes}, \sigma_{his-var}, \sigma_{ctf}) := \sigma_{name}$$

$$clocks(\sigma_{name}, \sigma_{svar}, \sigma_{non-svar}, \sigma_{clocks}, \sigma_{nodes}, \sigma_{his-var}, \sigma_{ctf}) := \sigma_{clocks}$$

$$clocks(\sigma_{name}, \sigma_{svar}, \sigma_{clocks}, \sigma_{nodes}) := \sigma_{clocks}$$

$$svar(\sigma_{name}, \sigma_{svar}, \sigma_{non-svar}, \sigma_{clocks}, \sigma_{nodes}, \sigma_{his-var}, \sigma_{ctf}) := \sigma_{svar}$$

$$svar(\sigma_{name}, \sigma_{svar}, \sigma_{clocks}, \sigma_{nodes}) := \sigma_{svar}$$

$$non-svar(\sigma_{name}, \sigma_{svar}, \sigma_{non-svar}, \sigma_{clocks}, \sigma_{nodes}, \sigma_{his-var}, \sigma_{ctf}) := \sigma_{non-svar}$$

$$nodes(\sigma_{name}, \sigma_{svar}, \sigma_{non-svar}, \sigma_{clocks}, \sigma_{nodes}, \sigma_{his-var}, \sigma_{ctf}) := \sigma_{nodes}$$

$$hvar(\sigma_{name}, \sigma_{svar}, \sigma_{non-svar}, \sigma_{clocks}, \sigma_{nodes}, \sigma_{his-var}, \sigma_{ctf}) := \sigma_{his-var}$$

$$ctf(\sigma_{name}, \sigma_{svar}, \sigma_{non-svar}, \sigma_{clocks}, \sigma_{nodes}, \sigma_{his-var}, \sigma_{ctf}) := \sigma_{ctf}$$

$$state(\sigma_{name}, \sigma_{svar}, \sigma_{non-svar}, \sigma_{clocks}, \sigma_{nodes}, \sigma_{his-var}, \sigma_{ctf}) := (\sigma_{name}, \sigma_{svar}, \sigma_{clocks}, \sigma_{nodes})$$

$$prepare(\sigma_{name}, \sigma_{svar}, \sigma_{clocks}, \sigma_{nodes}) := (\sigma_{name}, \sigma_{svar}, \sigma_{init-non-svar(\sigma_{name})}, \sigma_{clocks}, \sigma_{nodes}, \sigma_{svar}, false)$$

$$\frac{}{\sigma \rightarrow_{state} \sigma'}\ state(\sigma) = \sigma'$$

$$\frac{}{\sigma \rightarrow_{prepare} \sigma'}\ prepare(\sigma) = \sigma'$$

$$\mathrm{dom}(rel) := \text{domain of rel}$$

Note: AVL trees are represented as relations. So the domain is the keyset.

$$\sigma \lessdot\!+_{var} \{x \mapsto V\} := \begin{cases} (node(\sigma), svar(\sigma) \lessdot\!+ \{x \mapsto V\}, non-svar(\sigma), clocks(\sigma), nodes(\sigma), hvar(\sigma), ctf(\sigma)), & x \mapsto V \in svar(\sigma) \\ (node(\sigma), svar(\sigma), non-svar(\sigma) \lessdot\!+ \{x \mapsto V\}, clocks(\sigma), nodes(\sigma), hvar(\sigma), ctf(\sigma)), & x \mapsto V \notin svar(\sigma) \end{cases}$$

$$\sigma \lessdot\!+_{clock} \{c \mapsto V\} := (node(\sigma), svar(\sigma), non-svar(\sigma), clocks(\sigma) \lessdot\!+ \{c \mapsto V\}, nodes(\sigma), hvar(\sigma), ctf(\sigma))$$

$$\sigma \lessdot\!+_{node} \{id \mapsto \sigma_{node}\} := (node(\sigma), svar(\sigma), non-svar(\sigma), clocks(\sigma), nodes(\sigma) \lessdot\!+ \{id \mapsto \sigma_{node}\}, hvar(\sigma), ctf(\sigma))$$

$$\sigma \lessdot\!+_{ctf} val := (node(\sigma), svar(\sigma), non-svar(\sigma), clocks(\sigma), nodes(\sigma), hvar(\sigma), val)$$

$$x \mapsto V \in_{var} \sigma := x \mapsto V \in svar(\sigma) \cup non-svar(\sigma)$$

$$x \mapsto V \in_{svar} \sigma := x \mapsto V \in svar(\sigma)$$

$$c \mapsto V \in_{cl} \sigma := c \mapsto V \in clocks(\sigma)$$

$$x \mapsto \sigma_{node} \in_{node} \sigma := x \mapsto \sigma_{node} \in nodes(\sigma)$$

$$x \mapsto V \in_{his} \sigma := x \mapsto V \in hvar(\sigma)$$

C.2 Initialization

$$\frac{\sigma_{empty} \rightarrow_{init\_nodes(s_1;\ldots,s_n;)} \sigma_{init-nodes}}{\sigma_{root} \rightarrow_{init(\textbf{node}\ nodec\ldots;\ \textbf{let}\ s_1;\ldots,s_n;\ \textbf{tel};)} (nodec, svar_{nodec} \times \{nil\}, clocks_{nodec} \times \{true\}, \sigma_{init-nodes})}$$

$$\sigma_{init-non-svar(node)} := non-svar_{node} \times \{nil\}$$

$$\frac{\sigma \rightarrow_{init-nodes(s_1;)} \sigma', \ldots, \sigma^{(n-1)} \rightarrow_{init-nodes(s_n;)} \sigma^{(n)}}{\sigma \rightarrow_{init-nodes(s_1;\ldots;s_n;)} \sigma^{(n)}}$$

$$\frac{\sigma \rightarrow_{init-nodes(E)} \sigma'}{\sigma \rightarrow_{init-nodes(v_1,\ldots,v_n=E;)} \sigma'}$$

$$\frac{\sigma \rightarrow_{init-nodes(E)} \sigma'}{\sigma \rightarrow_{init-nodes(\textbf{assert}\ E;)} \sigma'}$$

$$\frac{}{\sigma \rightarrow_{init-nodes(E)} \sigma}\ literal(E)$$

$$\frac{}{\sigma \rightarrow_{init-nodes(E)} \sigma}\ variable(E)$$

$$\frac{\sigma \rightarrow_{init-nodes(E)} \sigma'}{\sigma \rightarrow_{init-nodes(op\ E)} \sigma'}$$

$$\frac{\sigma \rightarrow_{init-nodes(E_1)} \sigma', \sigma' \rightarrow_{init-nodes(E_2)} \sigma''}{\sigma \rightarrow_{init-nodes(E_1\ op\ E_2)} \sigma''}$$

$$\frac{\sigma \rightarrow_{init-nodes(C)} \sigma', \sigma' \rightarrow_{init-nodes(E_1)} \sigma'', \sigma'' \rightarrow_{init-nodes(E_2)} \sigma'''}{\sigma \rightarrow_{init-nodes(\textbf{if}\ C\ \textbf{then}\ E_1\ \textbf{else}\ E_2)} \sigma'''}$$

$$\frac{\sigma \rightarrow_{init-nodes(p_1)} \sigma', \ldots, \sigma^{(n-1)} \rightarrow_{init-nodes(p_n)} \sigma^{(n)}}{\sigma \rightarrow_{init-nodes(p_1,\ldots,p_n)} \sigma^{(n)}}$$

$$\frac{\sigma \rightarrow_{init-nodes(p_1,\ldots,p_n)} \sigma', \sigma_{root} \rightarrow_{init(ast)} \sigma_{init}}{\sigma \rightarrow_{init-nodes(nodec(p_1,\ldots,p_n))} (\sigma' \lhd\!\!\!+_{node} \{NID \mapsto \sigma_{init}\})}\ id(nodec(p_1,\ldots,p_n)) = NID, ast(nodec) = ast$$

C.3 Interpretation

C.3.1 Node

$$\frac{\sigma_{root} \rightarrow_{init(ast)} \sigma_{init}}{\sigma_{root} \rightarrow_{choose\_node(node)} \sigma_{init}} \; ast(node) = ast$$

$$\frac{\sigma_{state} \rightarrow_{sim(ast|p_1,\ldots,p_n|v_1,\ldots,v_n)} \sigma'_{state}}{\sigma_{state} \rightarrow_{clock\_step([p_1=v_1,\ldots,p_n=v_n])} \sigma'_{state}} \quad \begin{array}{c} node(\sigma_{state}) = nodec, ast(nodec) = ast, \\ parameters(ast) = p_1, \ldots, p_n, \\ \sigma'_{state} \neq \sigma_{cut-off} \end{array}$$

$$\frac{\begin{array}{c}\sigma_{state} \rightarrow_{prepare} \sigma', \sigma' \rightarrow_{sim-params(p_1,\ldots,p_n|v_1,\ldots,v_n)} \sigma'', \\ \sigma'' \rightarrow_{sim-body(s_1;\ldots;s_m;)} \sigma''', \sigma''' \rightarrow_{sim-clocks} \sigma'''', \sigma'''' \rightarrow_{state} \sigma'_{state}\end{array}}{\sigma_{state} \rightarrow_{sim(\textbf{node}\ nodec\ \ldots\ \textbf{let}\ s_1;\ldots;s_m;\ \textbf{tel};|p_1,\ldots,p_n|v_1,\ldots,v_n)} \sigma'_{state}} \; ctf(\sigma''') = false$$

$$\frac{\sigma_{state} \rightarrow_{prepare} \sigma', \sigma' \rightarrow_{sim-params(p_1,\ldots,p_n|v_1,\ldots,v_n)} \sigma'', \sigma'' \rightarrow_{sim-body(s_1;\ldots;s_m;)} \sigma'''}{\sigma_{state} \rightarrow_{sim(\textbf{node}\ nodec\ \ldots\ \textbf{let}\ s_1;\ldots;s_m;\ \textbf{tel};|p_1,\ldots,p_n|v_1,\ldots,v_n)} \sigma_{cut-off}} \; ctf(\sigma''') = true$$

$$\frac{\sigma_{state} \rightarrow_{prepare} \sigma', \sigma' \rightarrow_{sim-params(p_1,\ldots,p_n|v_1,\ldots,v_n)} \sigma'', \sigma'' \rightarrow_{sim-body(s_1;\ldots;s_m;)} \sigma_{abort}}{\sigma_{state} \rightarrow_{sim(\textbf{node}\ nodec\ \ldots\ \textbf{let}\ s_1;\ldots;s_m;\ \textbf{tel};|p_1,\ldots,p_n|v_1,\ldots,v_n)} \sigma_{abort}}$$

$$\frac{\sigma_{state} \rightarrow_{prepare} \sigma', \sigma' \rightarrow_{sim-params(p_1,\ldots,p_n|v_1,\ldots,v_n)} \sigma_{abort}}{\sigma_{state} \rightarrow_{sim(\textbf{node}\ nodec\ \ldots\ \textbf{let}\ s_1;\ldots;s_m;\ \textbf{tel};|p_1,\ldots,p_n|v_1,\ldots,v_n)} \sigma_{abort}}$$

$$\frac{\sigma \rightarrow_{update-param(p_1,v_1)} \sigma', \ldots, \sigma^{(n-1)} \rightarrow_{update-param(p_n,v_n)} \sigma^{(n)}}{\sigma \rightarrow_{sim-params(p_1,\ldots,p_n|v_1,\ldots,v_n)} \sigma^{(n)}}$$

$$\frac{\ldots, \sigma^{(i)} \rightarrow_{update-param(p_{i+1},v_{i+1})} \sigma_{abort}}{\sigma \rightarrow_{sim-params(p_1,\ldots,p_n|v_1,\ldots,v_n)} \sigma_{abort}}$$

$$\frac{\sigma \rightsquigarrow_{s_1;} \sigma', \ldots, \sigma^{(n-1)} \rightsquigarrow_{s_n;} \sigma^{(n)}}{\sigma \rightarrow_{sim-body(s_1;\ldots;s_n;)} \sigma^{(n)}}$$

$$\frac{\ldots \sigma^{(i)} \rightsquigarrow_{s_{i+1};} \sigma_{abort}}{\sigma \rightarrow_{sim-body(s_1;\ldots;s_n;)} \sigma_{abort}}$$

$$\frac{\sigma \rightarrow_{sim-clock(c_1)} \sigma', \ldots, \sigma^{(n-1)} \rightsquigarrow_{sim-clock(c_n)} \sigma^{(n)}}{\sigma \rightarrow_{sim-clocks} \sigma^{(n)}} \; \mathrm{dom}(clocks(\sigma)) = \{c_1, \ldots, c_n\}$$

$$\frac{\langle c, \sigma \rangle \Rightarrow_{active} true}{\sigma \rightarrow_{sim-clock(c)} (\sigma \lhd_{clock} \{c \mapsto false\})}$$

$$\frac{\langle c, \sigma \rangle \Rightarrow_{active} false}{\sigma \rightarrow_{sim-clock(c)} \sigma}$$

C.3.2 Equation

$$\frac{\sigma \rightarrow_{sim(E)} \sigma', \langle E, \sigma' \rangle \Rightarrow (e_1, \ldots, e_n), \sigma' \rightarrow_{update(v_1,e_1)} \sigma'', \ldots, \sigma^{(n)} \rightarrow_{update(v_n,e_n)} \sigma^{(n+1)}}{\sigma \rightsquigarrow_{v_1,\ldots,v_n=E;} \sigma^{(n+1)}}$$

$$\frac{\sigma \rightarrow_{sim(E)} \sigma', \langle E, \sigma' \rangle \Rightarrow (e_1, \ldots, e_n), \ldots, \sigma^{(i)} \rightarrow_{update(v_i,e_i)} \sigma_{abort}}{\sigma \rightsquigarrow_{v_1,\ldots,v_n=E;} \sigma_{abort}}$$

$$\frac{\sigma \rightarrow_{sim(E)} \sigma_{abort}}{\sigma \rightsquigarrow_{v_1,\ldots,v_n=E;} \sigma_{abort}}$$

C.3.3 Assertion

$$\frac{\sigma \rightarrow_{sim(E)} \sigma', \langle E, \sigma' \rangle \Rightarrow v}{\sigma \rightsquigarrow_{\textsf{assert}\ E;} (\sigma' \lhd_{ctf} c)} \; v \neq nil, c = (ctf(\sigma') = true \vee v = false)$$

$$\frac{\sigma \rightarrow_{sim(E)} \sigma', \langle E, \sigma' \rangle \Rightarrow nil}{\sigma \rightsquigarrow_{\textsf{assert}\ E;} \sigma_{abort}}$$

$$\frac{\sigma \rightarrow_{sim(E)} \sigma_{abort}}{\sigma \rightsquigarrow_{\textsf{assert}\ E;} \sigma_{abort}}$$

C.3.4 Update

$$\frac{\langle c, \sigma \rangle \Rightarrow_{active} true, \langle v, e, \sigma \rangle \Rightarrow_{c-nil} false}{\sigma \rightarrow_{update(v,e)} (\sigma \lhd_{var} \{v \mapsto e\})} \; cl(v) = c$$

$$\frac{\langle c, \sigma \rangle \Rightarrow_{active} true, \langle v, e, \sigma \rangle \Rightarrow_{c-nil} true}{\sigma \rightarrow_{update(v,e)} \sigma_{abort}} \; cl(v) = c$$

$$\frac{\langle c, \sigma \rangle \Rightarrow_{active} false}{\sigma \rightarrow_{update(v,e)} \sigma} \; cl(v) = c$$

$$\frac{\langle c, \sigma \rangle \Rightarrow_{active} true}{\sigma \rightarrow_{update-param(v,e)} (\sigma \lessdot_{var} \{v \mapsto e\})} \; main\_node(node(\sigma)),\; cl(v) = c$$

$$\frac{\langle c, \sigma \rangle \Rightarrow_{active} false}{\sigma \rightarrow_{update-param(v,e)} \sigma} \; main\_node(node(\sigma)),\; cl(v) = c$$

$$\frac{\sigma \rightarrow_{update(v,e)} \sigma'}{\sigma \rightarrow_{update-param(v,e)} \sigma'} \; \neg main\_node(node(\sigma))$$

$$\frac{\sigma \rightarrow_{update(v,e)} \sigma_{abort}}{\sigma \rightarrow_{update-param(v,e)} \sigma_{abort}} \; \neg main\_node(node(\sigma))$$

C.3.5 Expression

$$\frac{}{\sigma \rightarrow_{sim(E)} \sigma} \; literal(E)$$

$$\frac{}{\sigma \rightarrow_{sim(E)} \sigma} \; variable(E)$$

$$\frac{\sigma \rightarrow_{sim(E)} \sigma'}{\sigma \rightarrow_{sim(op\ E)} \sigma'}$$

$$\frac{\sigma \rightarrow_{sim(E)} \sigma_{abort}}{\sigma \rightarrow_{sim(op\ E)} \sigma_{abort}}$$

$$\frac{\sigma \rightarrow_{sim(E_1)} \sigma', \sigma' \rightarrow_{sim(E_2)} \sigma''}{\sigma \rightarrow_{sim(E_1\ op\ E_2)} \sigma''}$$

$$\frac{\sigma \rightarrow_{sim(E_1)} \sigma_{abort}}{\sigma \rightarrow_{sim(E_1\ op\ E_2)} \sigma_{abort}}$$

$$\frac{\sigma \rightarrow_{sim(E_1)} \sigma', \sigma' \rightarrow_{sim(E_2)} \sigma_{abort}}{\sigma \rightarrow_{sim(E_1\ op\ E_2)} \sigma_{abort}}$$

$$\frac{\sigma \rightarrow_{sim(C)} \sigma', \sigma' \rightarrow_{sim(E_1)} \sigma'', \sigma'' \rightarrow_{sim(E_2)} \sigma'''}{\sigma \rightarrow_{sim(\textbf{if}\ C\ \textbf{then}\ E_1\ \textbf{else}\ E_2)} \sigma'''}$$

$$\frac{\sigma \rightarrow_{sim(C)} \sigma_{abort}}{\sigma \rightarrow_{sim(\textbf{if } C \textbf{ then } E_1 \textbf{ else } E_2)} \sigma_{abort}}$$

$$\frac{\sigma \rightarrow_{sim(C)} \sigma', \sigma' \rightarrow_{sim(E_1)} \sigma_{abort}}{\sigma \rightarrow_{sim(\textbf{if } C \textbf{ then } E_1 \textbf{ else } E_2)} \sigma_{abort}}$$

$$\frac{\sigma \rightarrow_{sim(C)} \sigma', \sigma' \rightarrow_{sim(E_1)} \sigma'', \sigma'' \rightarrow_{sim(E_2)} \sigma_{abort}}{\sigma \rightarrow_{sim(\textbf{if } C \textbf{ then } E_1 \textbf{ else } E_2)} \sigma_{abort}}$$

$$\frac{\sigma \rightarrow_{sim(E_1)} \sigma', \ldots, \sigma^{(n-1)} \rightarrow_{sim(E_n)} \sigma^{(n)}}{\sigma \rightarrow_{sim(E_1,\ \ldots,\ E_n)} \sigma^{(n)}}$$

$$\frac{\ldots, \sigma^{(i)} \rightarrow_{sim(E_{i+1})} \sigma_{abort}}{\sigma \rightarrow_{sim(E_1,\ \ldots,\ E_n)} \sigma_{abort}}$$

$$\frac{\begin{array}{c}\sigma \rightarrow_{sim(p_1,\ldots,p_n)} \sigma', \langle ck, \sigma' \rangle \Rightarrow_{active} true, \\ \langle (p_1, \ldots, p_n), \sigma' \rangle \Rightarrow (v_1, \ldots, v_n), \\ \sigma_{node} \rightarrow_{sim(ast|i_1,\ldots,i_n|v_1,\ldots,v_n)} \sigma'_{node}\end{array}}{\sigma \rightarrow_{sim(nodec(p_1,\ldots,p_n))} ((\sigma' \mathbin{\triangleleft\!\!\!+}_{ctf} c) \mathbin{\triangleleft\!\!\!+}_{node} \{NID \mapsto \sigma'_{node}\})} \quad \begin{array}{c} cl(p_1) = ck, ast(nodec) = ast, \\ parameters(ast) = (i_1, \ldots, i_n), \\ id(nodec(p_1, \ldots, p_n)) = NID, \\ NID \mapsto \sigma_{node} \in_{node} \sigma', \\ c = (ctf(\sigma') = true \ \vee \ \sigma'_{node} = \sigma_{cut-off}) \end{array}$$

$$\frac{\begin{array}{c}\sigma \rightarrow_{sim(p_1,\ldots,p_n)} \sigma', \langle ck, \sigma' \rangle \Rightarrow_{active} true, \\ \langle (p_1, \ldots, p_n), \sigma' \rangle \Rightarrow (v_1, \ldots, v_n), \sigma_{node} \rightarrow_{sim(ast|i_1,\ldots,i_n|v_1,\ldots,v_n)} \sigma_{abort}\end{array}}{\sigma \rightarrow_{sim(nodec(p_1,\ldots,p_n))} \sigma_{abort}} \quad \begin{array}{c} cl(p_1) = ck, ast(nodec) = ast, \\ parameters(ast) = (i_1, \ldots, i_n), \\ id(nodec(p_1, \ldots, p_n)) = NID, \\ NID \mapsto \sigma_{node} \in_{node} \sigma' \end{array}$$

$$\frac{\sigma \rightarrow_{sim(p_1,\ldots,p_n)} \sigma', \langle ck, \sigma' \rangle \Rightarrow_{active} false}{\sigma \rightarrow_{sim(node(p_1,\ldots,p_n))} \sigma'} \quad cl(p_1) = ck$$

$$\frac{\sigma \rightarrow_{sim(p_1,\ldots,p_n)} \sigma_{abort}}{\sigma \rightarrow_{sim(node(p_1,\ldots,p_n))} \sigma_{abort}}$$

C.4 Evaluation

C.4.1 Literal and Identifier Expression

$$\frac{}{\langle x, \sigma \rangle \Rightarrow x}\ literal(x)$$

$$\frac{}{\langle x, \sigma \rangle \Rightarrow V}\ x \mapsto V \in_{var} \sigma$$

C.4.2 Data Expression

Unary Expressions

$$\frac{\langle E, \sigma \rangle \Rightarrow V}{\langle op\ E, \sigma \rangle \Rightarrow op\ V}\ V \neq nil, op \in \{-, not\}$$

$$\frac{\langle E, \sigma \rangle \Rightarrow nil}{\langle op\ E, \sigma \rangle \Rightarrow nil}\ op \in \{-, not\}$$

Binary Expressions

$$\frac{\langle E_1, \sigma \rangle \Rightarrow V_1, \langle E_2, \sigma \rangle \Rightarrow V_2}{\langle E_1\ op\ E_2, \sigma \rangle \Rightarrow V_1\ op\ V_2}\ V_1 \neq nil, V_2 \neq nil, op \in \{<, \leq, >, \geq, =, <>, +, -, *\}$$

$$\frac{\langle E_1, \sigma \rangle \Rightarrow V_1, \langle E_2, \sigma \rangle \Rightarrow V_2}{\langle E_1\ op\ E_2, \sigma \rangle \Rightarrow nil}\ (V_1 = nil \vee V_2 = nil), op \in \{<, \leq, >, \geq, =, <>, +, -, *\}$$

$$\frac{\langle E_1, \sigma \rangle \Rightarrow V_1, \langle E_2, \sigma \rangle \Rightarrow V_2}{\langle E_1/E_2, \sigma \rangle \Rightarrow \lfloor V_1/V_2 \rfloor}\ V_1 \neq nil, V_2 \neq nil, V_2 \neq 0, \tau(E_1/E_2) = int$$

$$\frac{\langle E_1, \sigma \rangle \Rightarrow V_1, \langle E_2, \sigma \rangle \Rightarrow V_2}{\langle E_1/E_2, \sigma \rangle \Rightarrow V_1/V_2}\ V_1 \neq nil, V_2 \neq nil, V_2 \neq 0.0, \tau(E_1/E_2) = real$$

$$\frac{\langle E_1, \sigma \rangle \Rightarrow V_1, \langle E_2, \sigma \rangle \Rightarrow V_2}{\langle E_1/E_2, \sigma \rangle \Rightarrow nil}\ (V_1 = nil \vee V_2 = nil \vee V_2 = 0), \tau(E_1/E_2) = int$$

$$\frac{\langle E_1, \sigma \rangle \Rightarrow V_1, \langle E_2, \sigma \rangle \Rightarrow V_2}{\langle E_1/E_2, \sigma \rangle \Rightarrow nil}\ (V_1 = nil \vee V_2 = nil \vee V_2 = 0.0), \tau(E_1/E_2) = real$$

$$\frac{\langle E_1,\sigma\rangle \Rightarrow V_1, \langle E_2,\sigma\rangle \Rightarrow V_2}{\langle E_1 \ \% \ E_2,\sigma\rangle \Rightarrow \mid V_1 \mid mod \mid V_2 \mid} \ V_1 \neq nil, V_1 \geq 0, V_2 \neq nil, V_2 \neq 0$$

$$\frac{\langle E_1,\sigma\rangle \Rightarrow V_1, \langle E_2,\sigma\rangle \Rightarrow V_2}{\langle E_1 \ \% \ E_2,\sigma\rangle \Rightarrow -(\mid V_1 \mid mod \mid V_2 \mid)} \ V_1 \neq nil, V_1 < 0, V_2 \neq nil, V_2 \neq 0$$

$$\frac{\langle E_1,\sigma\rangle \Rightarrow V_1, \langle E_2,\sigma\rangle \Rightarrow V_2}{\langle E_1 \ \% \ E_2,\sigma\rangle \Rightarrow nil} \ (V_1 = nil \vee V_2 = nil \vee V_2 = 0)$$

$$\frac{\langle E_1,\sigma\rangle \Rightarrow V_1, \langle E_2,\sigma\rangle \Rightarrow V_2}{\langle E_1 \ \textbf{or} \ E_2,\sigma\rangle \Rightarrow true} \ (V_1 = true \ \vee \ V_2 = true)$$

$$\frac{\langle E_1,\sigma\rangle \Rightarrow V_1, \langle E_2,\sigma\rangle \Rightarrow V_2}{\langle E_1 \ \textbf{or} \ E_2,\sigma\rangle \Rightarrow false} \ V_1 = false, V_2 = false$$

$$\frac{\langle E_1,\sigma\rangle \Rightarrow V_1, \langle E_2,\sigma\rangle \Rightarrow V_2}{\langle E_1 \ \textbf{or} \ E_2,\sigma\rangle \Rightarrow nil} \ V_1 \neq true, V_2 \neq true, (V_1 = nil \ \vee \ V_2 = nil)$$

$$\frac{\langle E_1,\sigma\rangle \Rightarrow V_1, \langle E_2,\sigma\rangle \Rightarrow V_2}{\langle E_1 \ \textbf{and} \ E_2,\sigma\rangle \Rightarrow false} \ (V_1 = false \ \vee \ V_2 = false)$$

$$\frac{\langle E_1,\sigma\rangle \Rightarrow V_1, \langle E_2,\sigma\rangle \Rightarrow V_2}{\langle E_1 \ \textbf{and} \ E_2,\sigma\rangle \Rightarrow true} \ V_1 = true, V_2 = true$$

$$\frac{\langle E_1,\sigma\rangle \Rightarrow V_1, \langle E_2,\sigma\rangle \Rightarrow V_2}{\langle E_1 \ \textbf{and} \ E_2,\sigma\rangle \Rightarrow nil} \ V_1 \neq false, V_2 \neq false, (V_1 = nil \ \vee \ V_2 = nil)$$

If-Then-Else Expressions

$$\frac{\langle C,\sigma\rangle \Rightarrow nil}{\langle \textbf{if} \ C \ \textbf{then} \ E_1 \ \textbf{else} \ E_2,\sigma\rangle \Rightarrow nil}$$

$$\frac{\langle C,\sigma\rangle \Rightarrow true, \langle E_1,\sigma\rangle \Rightarrow V}{\langle \textbf{if} \ C \ \textbf{then} \ E_1 \ \textbf{else} \ E_2,\sigma\rangle \Rightarrow V}$$

$$\frac{\langle C,\sigma\rangle \Rightarrow false, \langle E_2,\sigma\rangle \Rightarrow V}{\langle \textbf{if} \ C \ \textbf{then} \ E_1 \ \textbf{else} \ E_2,\sigma\rangle \Rightarrow V}$$

C.4.3 Temporal Expressions

$$\frac{}{\langle \textbf{pre}(E), \sigma\rangle \Rightarrow V}\ E \mapsto V \in_{his} \sigma$$

$$\frac{\langle E_1, \sigma\rangle \Rightarrow V}{\langle E_1 \texttt{->} E_2, \sigma\rangle \Rightarrow V}\ cl(E_1 \rightarrow E_2) = C, C \mapsto true \in_{cl} \sigma$$

$$\frac{\langle E_2, \sigma\rangle \Rightarrow V}{\langle E_1 \texttt{->} E_2, \sigma\rangle \Rightarrow V}\ cl(E_1 \rightarrow E_2) = C, C \mapsto false \in_{cl} \sigma$$

$$\frac{}{\langle E\ \textbf{when}\ C, \sigma\rangle \Rightarrow V}\ E \mapsto V \in_{var} \sigma$$

$$\frac{}{\langle \textbf{current}(E), \sigma\rangle \Rightarrow V}\ E \mapsto V \in_{var} \sigma$$

C.4.4 Node Calls

$$\frac{}{\langle node(p_1, \ldots, p_n), \sigma\rangle \Rightarrow (v_1, \ldots, v_m)}\ \begin{array}{c} output(node(p_1, \ldots, p_n)) = (o_1, \ldots, o_m), \\ id(node(p_1, \ldots, p_n)) = NID, \\ NID \mapsto \sigma_{node} \in_{node} \sigma, \\ o_1 \mapsto v_1 \in_{svar} \sigma_{node}, \ldots, \\ o_m \mapsto v_m \in_{svar} \sigma_{node} \end{array}$$

C.4.5 Clock Activeness

$$\frac{}{\langle c, \sigma\rangle \Rightarrow_{active} true}\ basic(c)$$

$$\frac{}{\langle c, \sigma\rangle \Rightarrow_{active} true}\ \begin{array}{c} \neg basic(c), c \mapsto true \in_{var} \sigma, \\ (\forall p.(p \in chierarchy_{node(\sigma)}\ \wedge\ p = c' \mapsto c)\ \Longrightarrow\ c' \mapsto true \in_{var} \sigma) \end{array}$$

$$\frac{}{\langle c, \sigma\rangle \Rightarrow_{active} false}\ \begin{array}{c} \neg basic(c), (c \mapsto false \in_{var} \sigma \vee \\ (\exists p.(p \in chierarchy_{node(\sigma)}\ \wedge\ p = c' \mapsto c)\ \Longrightarrow\ c' \mapsto false \in_{var} \sigma)) \end{array}$$

C.4.6 Verification

$$\frac{}{\langle v, e, \sigma\rangle \Rightarrow_{c-nil} true}\ v \in \mathrm{dom}(clocks(\sigma)), e = nil$$

$$\frac{}{\langle v, e, \sigma \rangle \Rightarrow_{c-nil} false} \; v \in \mathrm{dom}(clocks(\sigma)), e \neq nil$$

$$\frac{}{\langle v, e, \sigma \rangle \Rightarrow_{c-nil} false} \; v \notin \mathrm{dom}(clocks(\sigma))$$

$$\frac{}{\langle \sigma_{abort} \rangle \Rightarrow_{safe} false}$$

$$\frac{\langle \sigma_{state} \rangle \Rightarrow_{output-nil(ast)} p}{\langle \sigma_{state} \rangle \Rightarrow_{safe} \neg p} \quad \begin{array}{c} node(\sigma_{state}) = nodec, \\ ast(nodec) = ast, \neg verif\_node(nodec) \end{array}$$

$$\frac{\langle \sigma_{state} \rangle \Rightarrow_{output-nil(ast)} p_1, \langle \sigma_{state} \rangle \Rightarrow_{safe-prop(ast)} p_2}{\langle \sigma_{state} \rangle \Rightarrow_{safe} \neg p_1 \ \wedge \ p_2} \quad \begin{array}{c} node(\sigma_{state}) = nodec, \\ ast(nodec) = ast, verif\_node(nodec) \end{array}$$

$$\frac{\langle \sigma_{state} \rangle \Rightarrow_{output-nil(p_1)} false, \ldots, \langle \sigma_{state} \rangle \Rightarrow_{output-nil(p_n)} false}{\langle \sigma_{state} \rangle \Rightarrow_{output-nil(\textbf{node} \ \ldots \ \textbf{returns} \ (o_1;\ldots;o_n) \ \ldots)} false} \; oparameters(o_1; \ldots; o_n) = p_1, \ldots, p_n$$

$$\frac{\ldots, \langle \sigma_{state} \rangle \Rightarrow_{output-nil(p_i)} true}{\langle \sigma_{state} \rangle \Rightarrow_{output-nil(\textbf{node} \ \ldots \ \textbf{returns} \ (o_1;\ldots;o_n) \ \ldots)} true} \; oparameters(o_1; \ldots; o_n) = p_1, \ldots, p_n$$

$$\frac{}{\langle \sigma_{state} \rangle \Rightarrow_{output-nil(o)} true} \; clock(o) = c, o \mapsto nil \in_{svar} \sigma_{state}, c \mapsto false \in_{clock} \sigma_{state}$$

$$\frac{}{\langle \sigma_{state} \rangle \Rightarrow_{output-nil(o)} false} \; clock(o) = c, ((o \mapsto V \in_{svar} \sigma_{state} \wedge V \neq nil) \ \vee \ (c \mapsto true \in_{clock} \sigma_{state}))$$

$$\frac{}{\langle \sigma_{state} \rangle \Rightarrow_{safe-prop(\textbf{node} \ \ldots \ \textbf{returns}(o)\ldots)} true} \quad \begin{array}{c} oparameters(o) = p, \\ clock(p) = c, ((p \mapsto true \in_{svar} \sigma_{state}) \vee \\ (c \mapsto true \in_{clock} \sigma_{state})) \end{array}$$

$$\frac{}{\langle \sigma_{state} \rangle \Rightarrow_{safe-prop(\textbf{node} \ \ldots \ \textbf{returns}(o)\ldots)} false} \quad \begin{array}{c} oparameters(o) = p, \\ clock(p) = c, p \mapsto V \in_{svar} \sigma_{state}, \\ V \neq true, c \mapsto false \in_{clock} \sigma_{state} \end{array}$$

D Semantics of Translation to B

D.1 Auxiliary Definitions

$$format(str1, str2) = Replace\ \%s\ in\ str1\ by\ str2$$

$$machine\_name(node) = \ format(\text{“}M\_\%s\text{“}, node)$$

$$clock(E) = cl(E) = Clock\ of\ E$$

$$\tau(E) = Type\ of\ E$$

$$literal(E) := E\ is\ literal$$

$$variable(E) := E\ is\ variable$$

$$basic(C) := C\ is\ basic\ clock$$

$$main\_node(node) := node\ is\ main\ node$$

$$verif\_node(node) := node\ is\ verification\ node$$

$$id(node(p_1, \dots, p_n)) = ID\ of\ node$$

$$output(node(p_1, \dots, p_n)) = Output\ Variables\ of\ node$$

$$node\_name = Current\ Node\ Name$$

$$svar := state\ variables\ of\ current\ node$$

$$non - svar := non - state\ variables\ of\ current\ node$$

$$clocks := clocks\ of\ current\ node$$

$$chierarchy := clock\ hierarchy\ of\ current\ node$$

$$list(X) = X\ as\ List$$

$$reference\_name(id, node) = \ instance\_name(id)\text{“}.\text{“}machine\_name(node)$$

$$instance\_name(id) = \ format(\text{“}Node\_\%s\text{“}, id)$$

$$var\_to\_b(var) = \ format(\text{“}var\_\%s\text{“}, var)$$

$$param\_var\_to\_b(var) = \ format(\text{“}param\_\%s\text{“}, var)$$

$$hist\_var(var) = format(\text{“}hvar\_\%s\text{“}, var)$$

$$i - status(c) := \begin{cases} \text{“}is\_initialisation\text{“}, & c = \$basic \\ format(\text{“}is\_initialisation\_\%s\text{“}, c), & c \neq \$basic \end{cases}$$

$$def_{vars}(S) := \begin{cases} \varnothing, & S = \textbf{assert } E \\ \{v_1, \ldots, v_n\}, & S = v_1, \ldots, v_n = E \end{cases}$$

$$A \circ_{sep} B := \begin{cases} \epsilon, & A = \epsilon, B = \epsilon \\ A, & A \neq \epsilon, B = \epsilon \\ B, & A = \epsilon, B \neq \epsilon \\ A \; sep \; B, & A \neq \epsilon, B \neq \epsilon \end{cases}$$

$$\epsilon = \text{““} = Empty\ String$$

D.2 Node

$$[\![\textbf{node } nodec(input) \textbf{ returns}(output);\ \textbf{var } locals;\ \textbf{let } body \textbf{ tel};]\!]_{tr} =$$
$$\begin{cases} \textbf{MACHINE } machine - name \\ [\![body]\!]_{includes} \\ \textbf{SEES } LibraryLustre \\ \textbf{VARIABLES } VARS \\ \textbf{INVARIANT } [\![input, output, locals]\!]_{invariant} \\ \textbf{INITIALISATION } INIT \\ \textbf{OPERATIONS } [\![input, body]\!]_{clock-step} \\ \textbf{END} \end{cases} \text{if } \begin{array}{l} machine - name = machine\_name(nodec), \\ VARS = variables, \\ INIT = init \end{array}$$

D.3 Generation of INCLUDES

$$[\![s_1; \ldots; s_n;]\!]_{includes} = \epsilon \text{ if } [\![s_1;]\!]_{includes} = \epsilon, \ldots, [\![s_n;]\!]_{includes} = \epsilon$$

$$[\![s_1; \ldots; s_n;]\!]_{includes} = \textbf{INCLUDES } [\![s_1;]\!]_{includes} \circ_, \ldots \circ_, [\![s_n;]\!]_{includes}$$
$$\text{if } [\![s_1;]\!]_{includes} \neq \epsilon \text{ or } \ldots \text{ or } [\![s_n;]\!]_{includes} \neq \epsilon$$

$$[\![v_1, \ldots, v_n = E;]\!]_{includes} = [\![E]\!]_{includes}$$

$$[\![\textbf{assert } E;]\!]_{includes} = [\![E]\!]_{includes}$$

$$[\![E_1 \; op \; E_2]\!]_{includes} = [\![E_1]\!]_{includes} \circ_, [\![E_2]\!]_{includes}$$

$$[\![op \; E]\!]_{includes} = [\![E]\!]_{includes}$$

$$[\![\textbf{if } C \textbf{ then } E_1 \textbf{ else } E_2]\!]_{includes} = [\![C]\!]_{includes} \circ_, [\![E_1]\!]_{includes} \circ_, [\![E_2]\!]_{includes}$$

$$[\![var]\!]_{includes} = \epsilon \text{ if } variable(var)$$

$$[\![lit]\!]_{includes} = \epsilon \text{ if } literal(lit)$$

$$[\![E_1, \ldots, E_n]\!]_{includes} = [\![E_1]\!]_{includes} \circ_, \ldots \circ_, [\![E_n]\!]_{includes}$$

$$[\![node(p_1, \ldots, p_n)]\!]_{includes} = [\![p_1, \ldots, p_n]\!]_{includes} \circ_, ref$$
$$\text{if } nid = id(node(p_1, \ldots, p_n)),\ ref = reference\_name(nid, node)$$

D.4 Generation of VARIABLES

$$variables = \begin{cases} assert\_ok, \\ [\![c_1]\!]_{i-status}, \ldots, [\![c_n]\!]_{i-status}, \\ [\![s_1]\!]_{var}, \ldots, [\![s_n]\!]_{var} \end{cases} \text{if } \begin{array}{l} clocks = c_1, \ldots, c_n, \\ svar = s_1, \ldots, s_n \end{array}$$

$$[\![v]\!]_{var} = var \text{ if } var = var\_to\_b(v)$$

$$[\![c]\!]_{i-status} = var \text{ if } var = i - status(c)$$

D.5 Generation of INVARIANT

$$[\![input, output, locals]\!]_{invariant} = [\![input, output, locals]\!]_{typing} \text{ if } not(main\_node(node\_name))$$

$$[\![input, output, locals]\!]_{invariant} = \begin{cases} [\![input, output, locals]\!]_{typing} \ \& \\ [\![output]\!]_{not-nil} \end{cases} \text{if } main\_node(node\_name), not(verif\_node(node\_name))$$

$$[\![input, output, locals]\!]_{invariant} = \begin{cases} [\![input, output, locals]\!]_{typing} \ \& \\ [\![output]\!]_{not-nil} \ \& \\ [\![output]\!]_{safety} \end{cases} \text{if } verif\_node(node\_name)$$

D.5.1 Generation of Typing Predicate

$$[\![input, output, locals]\!]_{typing} = \begin{cases} assert\_ok \in \text{BOOL} \ \& \\ [\![c_1]\!]_{i-status} \in \text{BOOL} \\ \& \dots \& \\ [\![c_n]\!]_{i-status} \in \text{BOOL} \ \& \\ [\![input]\!]_{typing} \circ_{\&} [\![output]\!]_{typing} \circ_{\&} \\ [\![locals]\!]_{typing} \end{cases} \text{if } clocks = c_1, \dots, c_n$$

$$[\![d_1; \dots ; d_n]\!]_{typing} = [\![d_1]\!]_{typing} \circ_{\&} \dots \circ_{\&} [\![d_n]\!]_{typing}$$

$$[\![v_1, \dots, v_n : \tau]\!]_{typing} = [\![v_1, \tau]\!]_{typing-v} \circ_{\&} \dots \circ_{\&} [\![v_n, \tau]\!]_{typing-v}$$

$$[\![v_1, \dots, v_n : \tau \textbf{ when } C]\!]_{typing} = [\![v_1, \tau]\!]_{typing-v} \circ_{\&} \dots \circ_{\&} [\![v_n, \tau]\!]_{typing-v}$$

$$[\![v, \tau]\!]_{typing-v} = \epsilon \text{ if } v \notin svar$$

$$[\![v, \tau]\!]_{typing-v} = [\![v]\!]_{var} \in [\![\tau]\!]_{type} \text{ if } v \in svar$$

$$[\![\textbf{int}]\!]_{type} = LUSTRE\_INT$$

$$[\![\textbf{bool}]\!]_{type} = LUSTRE\_BOOL$$

D.5.2 Generation of Nil Checking and Safety Property

$$[\![d_1; \dots ; d_n]\!]_{not-nil} = [\![d_1]\!]_{not-nil} \ \& \ \dots \ \& \ [\![d_n]\!]_{not-nil}$$

$$[\![v_1, \dots, v_n : \tau]\!]_{not-nil} = \begin{cases} [\![\$basic]\!]_{i-status} = \textbf{FALSE} \Rightarrow [\![v_1]\!]_{var} \neq \varnothing \\ \& \dots \ \& \\ [\![\$basic]\!]_{i-status} = \textbf{FALSE} \Rightarrow [\![v_n]\!]_{var} \neq \varnothing \end{cases}$$

$$[\![v_1, \dots, v_n : \tau \textbf{ when } C]\!]_{not-nil} = \begin{cases} [\![C]\!]_{i-status} = \textbf{FALSE} \Rightarrow [\![v_1]\!]_{var} \neq \varnothing \\ \& \dots \ \& \\ [\![C]\!]_{i-status} = \textbf{FALSE} \Rightarrow [\![v_n]\!]_{var} \neq \varnothing \end{cases}$$

$$[\![v : \textbf{bool}]\!]_{safety} = [\![\$basic]\!]_{i-status} = \textbf{FALSE} \Rightarrow [\![v]\!]_{var} = \{ref \mapsto \textbf{TRUE}\}$$

$$[\![v : \textbf{bool when } C]\!]_{safety} = [\![C]\!]_{i-status} = \textbf{FALSE} \Rightarrow [\![v]\!]_{var} = \{ref \mapsto \textbf{TRUE}\}$$

D.6 Generation of INITIALISATION

$$init = \begin{cases} assert\_ok := \textbf{TRUE}; \\ [\![c_1]\!]_{i-status} := \textbf{TRUE}; \\ \ldots; \\ [\![c_n]\!]_{i-status} := \textbf{TRUE}; & \text{if } \begin{array}{l} clocks = c_1, \ldots, c_n, \\ svar = v_1, \ldots, v_n \end{array} \\ [\![v_1]\!]_{var} := \varnothing; \\ \ldots; \\ [\![v_n]\!]_{var} := \varnothing \end{cases}$$

D.7 Generation of Clock Step Operation

$$[\![input, body]\!]_{clock-step} = \begin{cases} clock\_step([\![input]\!]_{params}) := \\ \quad \textbf{PRE} \\ \quad\quad [\![input]\!]_{params-typing} \\ \quad \textbf{THEN} \\ \quad\quad [\![input]\!]_{params-assign}\circ; & \text{if } \begin{array}{l} [\![body]\!]_{op-locals} = \epsilon, \\ clocks = c_1 \ldots, c_n \end{array} \\ \quad\quad [\![body]\!]_{tr}\circ; \\ \quad\quad [\![body]\!]_{assert\_cut}\circ; \\ \quad\quad [\![c_1, \ldots, c_n]\!]_{clocks-update} \\ \quad \textbf{END} \end{cases}$$

$$[\![input, body]\!]_{clock-step} = \begin{cases} clock\_step([\![input]\!]_{params}) := \\ \quad \textbf{PRE} \\ \quad\quad [\![input]\!]_{params-typing} \\ \quad \textbf{THEN} \\ \quad\quad \textbf{VAR } [\![body]\!]_{op-locals} \textbf{ IN} \\ \quad\quad\quad [\![input]\!]_{params-assign}\circ; \\ \quad\quad\quad [\![body]\!]_{h-asn}\circ; & \text{if } \begin{array}{l} [\![body]\!]_{op-locals} \neq \epsilon, \\ clocks = c_1 \ldots, c_n \end{array} \\ \quad\quad\quad [\![body]\!]_{tr}\circ; \\ \quad\quad\quad [\![body]\!]_{assert\_cut}\circ; \\ \quad\quad\quad [\![c_1, \ldots, c_n]\!]_{clocks-update} \\ \quad\quad \textbf{END} \\ \quad \textbf{END} \end{cases}$$

D.7.1 Clock Step Parameters

$$[\![d_1; \ldots; d_n]\!]_{params} = [\![d_1]\!]_{params}, \ldots, [\![d_n]\!]_{params}$$

$$[\![v_1, \ldots, v_n : \tau]\!]_{params} = [\![v_1]\!]_{param}, \ldots, [\![v_n]\!]_{param}$$

$$[\![v_1, \ldots, v_n : \tau \textbf{ when } C]\!]_{params} = [\![v_1]\!]_{param}, \ldots, [\![v_n]\!]_{param}$$

$$[\![v]\!]_{param} = param \text{ if } param = param\_var\_to\_b(v)$$

$$[\![d_1; \ldots; d_n]\!]_{params-typing} = [\![d_1]\!]_{params-typing} \;\&\; \ldots \;\&\; [\![d_n]\!]_{params-typing}$$

$$[\![v_1, \ldots, v_n : \tau]\!]_{params-typing} = [\![v_1, \tau]\!]_{param-typing} \;\&\; \ldots \;\&\; [\![v_n, \tau]\!]_{param-typing}$$

$$[\![v_1, \ldots, v_n : \tau \textbf{ when } C]\!]_{params-typing} = [\![v_1, \tau]\!]_{param-typing} \;\&\; \ldots \;\&\; [\![v_n, \tau]\!]_{param-typing}$$

$$[\![v, \tau]\!]_{param-typing} = [\![v]\!]_{param} \in [\![\tau]\!]_{param-type}$$

$$[\![\textbf{int}]\!]_{param-type} = LUSTRE\_INT\_NOT\_NIL \text{ if } main\_node(node\_name)$$

$$[\![\textbf{int}]\!]_{param-type} = LUSTRE\_INT \text{ if } not(main\_node(node\_name))$$

$$[\![\textbf{bool}]\!]_{param-type} = LUSTRE\_BOOL\_NOT\_NIL \text{ if } main\_node(node\_name)$$

$$[\![\textbf{bool}]\!]_{param-type} = LUSTRE\_BOOL \text{ if } not(main\_node(node\_name))$$

$$[\![d_1; \ldots; d_n]\!]_{params-assign} = [\![d_1]\!]_{params-assign}; \ldots; [\![d_n]\!]_{params-assign}$$

$$[\![v_1, \ldots, v_n : \tau]\!]_{params-assign} = [\![v_1]\!]_{param-assign}; \ldots ; [\![v_n]\!]_{param-assign}$$

$$[\![v_1, \ldots, v_n : \tau \textbf{ when } C]\!]_{params-assign} = [\![v_1]\!]_{param-assign}; \ldots ; [\![v_n]\!]_{param-assign}$$

$$[\![v]\!]_{param-assign} = [\![v]\!]_{var} := [\![v]\!]_{param} \text{ if } clock(v) = \$basic, v \notin clocks$$

$$[\![v]\!]_{param-assign} = \begin{cases} [\![v]\!]_{var} := [\![v]\!]_{param}; \\ \textbf{PRE } [\![v]\!]_{var} \neq \varnothing \textbf{ THEN } skip \textbf{ END} \end{cases} \text{if } clock(v) = \$basic, v \in clocks$$

$$[\![v]\!]_{param-assign} = \begin{cases} \textbf{IF}[\![c]\!]_{active}\textbf{THEN} \\ \quad [\![v]\!]_{var} := [\![v]\!]_{param} \\ \textbf{END} \end{cases} \text{if } clock(v) \neq \$basic, v \notin clocks$$

$$[\![v]\!]_{param-assign} = \begin{cases} \textbf{IF}[\![c]\!]_{active}\textbf{THEN} \\ \quad [\![v]\!]_{var} := [\![v]\!]_{param}; \\ \quad \textbf{PRE } [\![v]\!]_{var} \neq \varnothing \textbf{ THEN } skip \textbf{ END} \\ \textbf{END} \end{cases} \text{if } clock(v) \neq \$basic, v \in clocks$$

D.7.2 Generation of Local Variables

$[\![body]\!]_{op-locals} = [\![body]\!]_{h-vars} \circ, [\![v_1]\!]_{var} \circ, \ldots \circ, [\![v_n]\!]_{var}$ if $non - svar = v_1, \ldots, v_n$

$[\![s_1; \ldots ; s_n;]\!]_{h-vars} = [\![s_1; , l_1]\!]_{h-vars} \circ, \ldots \circ, [\![s_n; , l_n]\!]_{h-vars}$
if $l_1 = \varnothing, l_2 = l_1 \cup def_{vars}(s_1;), \ldots, l_n = l_{n-1} \cup def_{vars}(s_{n-1};)$

$[\![v_1, \ldots, v_n = E; , dvars]\!]_{h-vars} = [\![E, dvars]\!]_{h-vars}$

$[\![\textbf{assert } E; , dvars]\!]_{h-vars} = [\![E, dvars]\!]_{h-vars}$

$[\![E_1 \; op \; E_2, dvars]\!]_{h-vars} = [\![E_1, dvars]\!]_{h-vars} \circ, [\![E_2, dvars]\!]_{h-vars}$

$[\![op \; E, dvars]\!]_{h-vars} = [\![E, dvars]\!]_{h-vars}$ if $op \neq \textbf{pre}$

$[\![op \; E, dvars]\!]_{h-vars} = \epsilon$ if $op = \textbf{pre}, E \notin dvars$

$[\![op \; E, dvars]\!]_{h-vars} = [\![E]\!]_{h-var}$ if $op = \textbf{pre}, E \in dvars$

$[\![\textbf{if } C \textbf{ then } E_1 \textbf{ else } E_2, dvars]\!]_{h-vars} = [\![C, dvars]\!]_{h-vars} \circ, [\![E_1, dvars]\!]_{h-vars} \circ, [\![E_2, dvars]\!]_{h-vars}$

$[\![var, dvars]\!]_{h-vars} = \epsilon$ if $variable(var)$

$[\![lit, dvars]\!]_{h-vars} = \epsilon$ if $literal(lit)$

$[\![node(p_1, \ldots, p_n), dvars]\!]_{h-vars} = [\![p_1, dvars]\!]_{h-vars} \circ, \ldots \circ, [\![p_n, dvars]\!]_{h-vars}$

$[\![s_1; \ldots ; s_n;]\!]_{h-asn} = [\![s_1; , l_1]\!]_{h-asn} \circ_; \ldots \circ_; [\![s_n; , l_n]\!]_{h-asn}$
if $l_1 = \varnothing, l_2 = l_1 \cup def_{vars}(s_1;), \ldots, l_n = l_{n-1} \cup def_{vars}(s_{n-1};)$

$[\![v_1, \ldots, v_n = E; , dvars]\!]_{h-asn} = [\![E, dvars]\!]_{h-asn}$

$[\![\textbf{assert } E; , dvars]\!]_{h-asn} = [\![E, dvars]\!]_{h-asn}$

$[\![E_1 \; op \; E_2, dvars]\!]_{h-asn} = [\![E_1, dvars]\!]_{h-asn} \circ_; [\![E_2, dvars]\!]_{h-asn}$

$[\![op \; E, dvars]\!]_{h-asn} = [\![E, dvars]\!]_{h-asn}$ if $op \neq \textbf{pre}$

$[\![op \; E, dvars]\!]_{h-asn} = \epsilon$ if $op = \textbf{pre}, E \notin dvars$

$[\![op \; E, dvars]\!]_{h-asn} = [\![E]\!]_{h-var} := [\![E]\!]_{var}$ if $op = \textbf{pre}, E \in dvars$

$[\![\textbf{if } C \textbf{ then } E_1 \textbf{ else } E_2, dvars]\!]_{h-asn} = [\![C, dvars]\!]_{h-asn} \circ_; [\![E_1, dvars]\!]_{h-asn} \circ_; [\![E_2, dvars]\!]_{h-asn}$

$[\![v, dvars]\!]_{h-asn} = \epsilon$ if $variable(v)$

$[\![lit, dvars]\!]_{h-asn} = \epsilon$ if $literal(lit)$

$[\![(E_1, \ldots, E_n), dvars]\!]_{h-asn} = [\![E_1, dvars]\!]_{h-asn} \circ_; \ldots \circ_; [\![E_n, dvars]\!]_{h-asn}$

$[\![node(p_1, \ldots, p_n), dvars]\!]_{h-asn} = [\![(p_1, \ldots, p_n), dvars]\!]_{h-asn}$

$[\![v]\!]_{h-var} = var$ if $var = hist\_var(v)$

D.7.3 Generation from Node Body

Node Body

$$[\![s_1; \ldots; s_n;]\!]_{tr} = [\![s_1;, l_1]\!]_{tr}; \ldots; [\![s_n;, l_n]\!]_{tr}$$
$$\text{if } l_1 = \varnothing, l_2 = l_1 \cup def_{vars}(s_1;), \ldots, l_n = l_{n-1} \cup def_{vars}(s_{n-1};)$$

Equation

$$[\![v_1, \ldots, v_n = E;, dvars]\!]_{tr} = \left\{ \begin{array}{l} [\![E, dvars]\!]_{node-sim} \circ; \\ [\![v_1, e_1]\!]_{eq}; \\ \ldots; \\ [\![v_n, e_n]\!]_{eq}; \end{array} \right. \quad \text{if } [\![E, dvars]\!]_{tr} = e_1, \ldots, e_n$$

$[\![v, e]\!]_{eq} = [\![v]\!]_{var} := e$ if $clock(v) = \$basic, v \notin clocks$

$$[\![v, e]\!]_{eq} = \left\{ \begin{array}{l} \textbf{IF } [\![c]\!]_{active} \textbf{ THEN} \\ \quad [\![v]\!]_{var} := e \\ \textbf{END} \end{array} \right. \quad \text{if } clock(v) \neq \$basic, v \notin clocks$$

$$[\![v, e]\!]_{eq} = \left\{ \begin{array}{l} [\![v]\!]_{var} := e; \\ \textbf{PRE } [\![v]\!]_{var} \neq \varnothing \textbf{ THEN } skip \textbf{ END} \end{array} \right. \quad \text{if } clock(v) = \$basic, v \in clocks$$

$$[\![v, e]\!]_{eq} = \left\{ \begin{array}{l} \textbf{IF}[\![c]\!]_{active}\textbf{THEN} \\ \quad [\![v]\!]_{var} := e; \\ \quad \textbf{PRE } [\![v]\!]_{var} \neq \varnothing \textbf{ THEN } skip \textbf{ END} \\ \textbf{END} \end{array} \right. \quad \text{if } clock(v) \neq \$basic, v \in clocks$$

Assertion

$$[\![\textbf{assert } E;, dvars]\!]_{tr} = \left\{ \begin{array}{l} [\![E, dvars]\!]_{node-sim} \circ; \\ \textbf{PRE } [\![E, dvars]\!]_{tr} \neq \varnothing \textbf{ THEN } skip \textbf{ END}; \\ assert\_ok := bool(assert\_ok = \textbf{TRUE} \ \& \ [\![E, dvars]\!]_{tr} = \{ref \mapsto \textbf{TRUE}\}) \end{array} \right.$$

$[\![body]\!]_{assert-cut} = \textbf{SELECT } [\![body]\!]_{assert-true} \textbf{ THEN } skip \textbf{ END}$ if $main\_node(node\_name)$

$[\![body]\!]_{assert-cut} = \epsilon$ if $not(main\_node(node\_name))$

$[\![s_1; \ldots; s_n;]\!]_{assert-true} = assert\_ok = \textbf{TRUE} \circ_{\&} [\![s_1;]\!]_{assert-true} \circ_{\&} \ldots \circ_{\&} [\![s_n;]\!]_{assert-true}$

$[\![v_1, \ldots, v_n = E;]\!]_{assert-true} = [\![E]\!]_{assert-true}$

$[\![\textbf{assert } E;]\!]_{assert-true} = [\![E]\!]_{assert-true}$

$[\![E_1 \ op \ E_2]\!]_{assert-true} = [\![E_1]\!]_{assert-true} \circ_{\&} [\![E_2]\!]_{assert-true}$

$[\![op \ E]\!]_{assert-true} = [\![E]\!]_{assert-true}$

$[\![\textbf{if } C \textbf{ then } E_1 \textbf{ else } E_2]\!]_{assert-true} = [\![C]\!]_{assert-true} \circ_{\&} [\![E_1]\!]_{assert-true} \circ_{\&} [\![E_2]\!]_{assert-true}$

$[\![var]\!]_{assert-true} = \epsilon$ if $variable(var)$

$[\![lit]\!]_{assert-true} = \epsilon$ if $literal(lit)$

$[\![E_1, \ldots, E_n]\!]_{assert-true} = [\![E_1]\!]_{assert-true} \circ_{\&} \ldots \circ_{\&} [\![E_n]\!]_{assert-true}$

$$[\![node(p_1, \ldots; p_n)]\!]_{assert-true} = \left\{ \begin{array}{l} [\![p_1, \ldots, p_n]\!]_{assert-true} \circ_{\&} \\ iname.assert\_ok = \textbf{TRUE} \end{array} \right. \text{if } \begin{array}{l} nid = id(node(p_1, \ldots, p_n)), \\ iname = instance\_name(nid) \end{array}$$

Node Call Simulation

$[\![E_1\ op\ E_2, dvars]\!]_{node-sim} = [\![E_1, dvars]\!]_{node-sim} \circ;\ [\![E_2, dvars]\!]_{node-sim}$

$[\![op\ E, dvars]\!]_{node-sim} = [\![E, dvars]\!]_{node-sim}$

$[\![\textbf{if}\ C\ \textbf{then}\ E_1\ \textbf{else}\ E_2, dvars]\!]_{node-sim} = [\![C, dvars]\!]_{node-sim} \circ;\ [\![E_1, dvars]\!]_{node-sim} \circ;\ [\![E_2, dvars]\!]_{node-sim}$

$[\![var, dvars]\!]_{node-sim} = \epsilon$ if $variable(var)$

$[\![lit, dvars]\!]_{node-sim} = \epsilon$ if $literal(lit)$

$[\![(E_1, \ldots, E_n), dvars]\!]_{node-sim} = [\![E_1, dvars]\!]_{node-sim} \circ;\ \ldots \circ;\ [\![E_n, dvars]\!]_{node-sim}$

$$[\![node(p_1, \ldots, p_n), dvars]\!]_{node-sim} = \left\{ \begin{array}{l} [\![(p_1, \ldots, p_n), dvars]\!]_{node-sim} \circ; \\ [\![nid]\!]_{clock-step-name}([\![(p_1, \ldots, p_n), dvars]\!]_{tr}) \end{array} \right.$$
if $nid = id(node(p_1, \ldots, p_n)),\ clock(p_1) = \$basic$

$$[\![node(p_1, \ldots, p_n), dvars]\!]_{node-sim} = \left\{ \begin{array}{l} [\![(p_1, \ldots, p_n), dvars]\!]_{node-sim} \circ; \\ \textbf{IF}\ [\![c]\!]_{active} \textbf{THEN} \\ \quad [\![nid]\!]_{clock-step-name}([\![(p_1, \ldots, p_n), dvars]\!]_{tr}) \\ \textbf{END} \end{array} \right.$$
if $nid = id(node(p_1, \ldots, p_n)),\ clock(p_1) = c, c \neq \$basic$

$[\![nid]\!]_{clock-step-name} = iname.clock\_step$ if $iname = instance\_name(nid)$

Data Expression

$[\![E_1 + E_2, dvars]\!]_{tr} = l\_plus([\![E_1, dvars]\!]_{tr} \mapsto [\![E_2, dvars]\!]_{tr})$

$[\![E_1 - E_2, dvars]\!]_{tr} = l\_minus([\![E_1, dvars]\!]_{tr} \mapsto [\![E_2, dvars]\!]_{tr})$

$[\![E_1 * E_2, dvars]\!]_{tr} = l\_multiply([\![E_1, dvars]\!]_{tr} \mapsto [\![E_2, dvars]\!]_{tr})$

$[\![E_1 / E_2, dvars]\!]_{tr} = l\_divide([\![E_1, dvars]\!]_{tr} \mapsto [\![E_2, dvars]\!]_{tr})$

$[\![E_1\ \textbf{div}\ E_2, dvars]\!]_{tr} = l\_divide([\![E_1, dvars]\!]_{tr} \mapsto [\![E_2, dvars]\!]_{tr})$

$[\![E_1\ \%\ E_2, dvars]\!]_{tr} = l\_modulo([\![E_1, dvars]\!]_{tr} \mapsto [\![E_2, dvars]\!]_{tr})$

$[\![E_1\ \textbf{mod}\ E_2, dvars]\!]_{tr} = l\_modulo([\![E_1, dvars]\!]_{tr} \mapsto [\![E_2, dvars]\!]_{tr})$

$[\![-E, dvars]\!]_{tr} = l\_unary\_minus([\![E, dvars]\!]_{tr})$

$[\![E_1 < E_2, dvars]\!]_{tr} = l\_less([\![E_1, dvars]\!]_{tr} \mapsto [\![E_2, dvars]\!]_{tr})$

$[\![E_1 <= E_2, dvars]\!]_{tr} = l\_less\_equal([\![E_1, dvars]\!]_{tr} \mapsto [\![E_2, dvars]\!]_{tr})$

$[\![E_1 > E_2, dvars]\!]_{tr} = l\_greater([\![E_1, dvars]\!]_{tr} \mapsto [\![E_2, dvars]\!]_{tr})$

$[\![E_1 >= E_2, dvars]\!]_{tr} = l\_greater\_equal([\![E_1, dvars]\!]_{tr} \mapsto [\![E_2, dvars]\!]_{tr})$

$[\![E_1 = E_2, dvars]\!]_{tr} = l\_equal\_integer([\![E_1, dvars]\!]_{tr} \mapsto [\![E_2, dvars]\!]_{tr})$ if $\tau(E_1 = E_2) = \textbf{int}$

$[\![E_1 = E_2, dvars]\!]_{tr} = l\_equal\_boolean([\![E_1, dvars]\!]_{tr} \mapsto [\![E_2, dvars]\!]_{tr})$ if $\tau(E_1 = E_2) = \textbf{bool}$

$[\![E_1 <> E_2, dvars]\!]_{tr} = l\_unequal\_integer([\![E_1, dvars]\!]_{tr} \mapsto [\![E_2, dvars]\!]_{tr})$ if $\tau(E_1 <> E_2) = \textbf{int}$

$[\![E_1 <> E_2, dvars]\!]_{tr} = l\_unequal\_boolean([\![E_1, dvars]\!]_{tr} \mapsto [\![E_2, dvars]\!]_{tr})$ if $\tau(E_1 <> E_2) = \textbf{bool}$

$[\![E_1\ \textbf{and}\ E_2, dvars]\!]_{tr} = l\_and([\![E_1, dvars]\!]_{tr} \mapsto [\![E_2, dvars]\!]_{tr})$

$[\![E_1\ \textbf{or}\ E_2, dvars]\!]_{tr} = l\_or([\![E_1, dvars]\!]_{tr} \mapsto [\![E_2, dvars]\!]_{tr})$

$[\![\textbf{not}\ E, dvars]\!]_{tr} = l\_not([\![E, dvars]\!]_{tr})$

$[\![\textbf{if } C \textbf{ then } E_1 \textbf{ else } E_2, dvars]\!]_{tr} = \mathit{l\_ite\_integer}([\![C, dvars]\!]_{tr} \mapsto [\![E_1, dvars]\!]_{tr} \mapsto [\![E_2, dvars]\!]_{tr})$
if $\tau(E_1) = \tau(E_2) = \textbf{int}$

$[\![\textbf{if } C \textbf{ then } E_1 \textbf{ else } E_2, dvars]\!]_{tr} = \mathit{l\_ite\_boolean}([\![C, dvars]\!]_{tr} \mapsto [\![E_1, dvars]\!]_{tr} \mapsto [\![E_2, dvars]\!]_{tr})$
if $\tau(E_1) = \tau(E_2) = \textbf{bool}$

$[\![v, dvars]\!]_{tr} = [\![v]\!]_{var}$ if $variable(v)$

$[\![lit, dvars]\!]_{tr} = \{ref \mapsto lit\}$ if $literal(lit)$

$[\![(E_1, \ldots, E_n), dvars]\!]_{tr} = [\![E_1, dvars]\!]_{tr}, \ldots, [\![E_n, dvars]\!]_{tr}$

$[\![node(p_1, \ldots, p_n), dvars]\!]_{tr} = [\![o_1, nid]\!]_{read-out}, \ldots, [\![o_m, nid]\!]_{read-out}$
if $id(node(p_1, \ldots, p_n)) = nid,\ output(node(p_1, \ldots, p_n)) = o_1, \ldots, o_m$

$[\![o, nid]\!]_{read-out} = \ iname.[\![o]\!]_{var}$ if $iname = instance\_name(nid)$

Temporal Expression

$[\![E_1 \texttt{ -> } E_2, dvars]\!]_{tr} = \mathit{l\_fby\_integer}([\![C]\!]_{i-status} \mapsto [\![E_1, dvars]\!]_{tr} \mapsto [\![E_2, dvars]\!]_{tr})$
if $clock(E_1 \texttt{ -> } E_2) = C, \tau(E_1) = \tau(E_2) = \textbf{int}$

$[\![E_1 \texttt{ -> } E_2, dvars]\!]_{tr} = \mathit{l\_fby\_boolean}([\![C]\!]_{i-status} \mapsto [\![E_1, dvars]\!]_{tr} \mapsto [\![E_2, dvars]\!]_{tr})$
if $clock(E_1 \texttt{ -> } E_2) = C, \tau(E_1) = \tau(E_2) = \textbf{bool}$

$[\![\textbf{pre } E, dvars]\!]_{tr} = [\![E]\!]_{var}$ if $E \notin dvars$

$[\![\textbf{pre } E, dvars]\!]_{tr} = [\![E]\!]_{h-var}$ if $E \in dvars$

$[\![\textbf{current } E, dvars]\!]_{tr} = [\![E]\!]_{var}$

$[\![E \textbf{ when } C, dvars]\!]_{tr} = [\![E]\!]_{var}$

Clock Activeness

$superclocks(c) = list(\{c' | c' \mapsto c : chierarchy\})$

$$[\![c]\!]_{active} = \begin{cases} [\![c_1]\!]_{var} = \{ref \mapsto \textbf{TRUE}\}\ \& \\ \ldots\ \& \\ [\![c_n]\!]_{var} = \{ref \mapsto \textbf{TRUE}\}\ \& \\ [\![c]\!]_{var} = \{ref \mapsto \textbf{TRUE}\} \end{cases} \text{if } superclocks(c) = c_1, \ldots, c_n$$

Clock Initialization Status

$[\![c_1, \ldots, c_n]\!]_{clocks-update} = [\![c_1]\!]_{clock-update}; \ldots ; [\![c_n]\!]_{clock-update}$

$[\![c]\!]_{clock-update} = [\![c]\!]_{i-status} := \textbf{FALSE}$ if $c = \$basic$

$$[\![c]\!]_{clock-update} = \begin{cases} \textbf{IF } [\![c]\!]_{active} \textbf{ THEN} \\ \quad [\![c]\!]_{i-status} := \textbf{FALSE} \\ \textbf{END} \end{cases} \text{if } c \neq \$basic$$

E Lustre Translations - Examples

Lift_verif.lus:

```
node Lift(up, down : bool) returns (exec_up, exec_down : bool; floor : int);
var old_floor : int;
let
    old_floor = 0 -> pre(floor);
    exec_up = up and old_floor < 100;
    exec_down = down and old_floor > 0;
    floor = if exec_up then old_floor + 1 else
              if exec_down then old_floor - 1 else old_floor;
tel;

node Lift_verif(up, down : bool) returns (ok : bool);
var exec_up, exec_down, only_exec_up_or_down,
    not_lower_than_lowest_floor, not_higher_than_highest_floor : bool;
    floor : int;
let
    exec_up, exec_down, floor = Lift(up,down);

    -- ASSERTIONS
    assert (not up and not down) -> not(up and down);


    -- PROPERTIES
    only_exec_up_or_down = not(exec_up and exec_down);
    not_lower_than_lowest_floor = floor >= 0;
    not_higher_than_highest_floor = floor <= 100;

    ok = only_exec_up_or_down and not_lower_than_lowest_floor and
        not_higher_than_highest_floor;
tel;
```

SLOW_TIME_STABLE.lus:

```
node SLOW_TIME_STABLE(set, second : bool; delay : int) returns (level : bool when second);
let
  level = SLOW_STABLE((set,delay) when second);
tel;

node SLOW_STABLE(set : bool; delay : int) returns (level : bool);
var count : int;
let
  level = (count > 0);
  count = if set then delay else if false -> pre(level) then pre(count) - 1 else 0;
tel;
```

UMS.lus:

```
-- taken from Programming and verifying real-time systems by means of the
-- synchronous data-flow language LUSTRE, N.Halbwachs et. al

node UMS(on_A, on_B, on_C, ack_AB, ack_BC : bool) returns (grant_access,
        grant_exit, do_AB, do_BC : bool);
var empty_section, only_on_B : bool;
let
    grant_access = empty_section and ack_AB;
    grant_exit = only_on_B and ack_BC;
    do_AB = not ack_AB and empty_section;
    do_BC = not ack_BC and only_on_B;
    empty_section = not(on_A or on_B or on_C);
    only_on_B = on_B and not(on_A or on_C);
tel;
```

Translation of Lift_verif.lus to B with Lift_verif as main node and verification node:

Lift_verif:

```
MACHINE M_Lift_verif
INCLUDES Node_1.M_Lift
SEES LibraryLustre

VARIABLES assert_ok, is_initialisation, var_ok, var_only_exec_up_or_down,
var_not_lower_than_lowest_floor, var_not_higher_than_highest_floor

INVARIANT ((assert_ok : BOOL) & ((is_initialisation : BOOL) & ((var_ok : LUSTRE_BOOL) &
((var_not_higher_than_highest_floor : LUSTRE_BOOL) &
((var_not_lower_than_lowest_floor : LUSTRE_BOOL) &
((var_only_exec_up_or_down : LUSTRE_BOOL) &
(((is_initialisation = FALSE) => (var_ok /= {})) &
((is_initialisation = FALSE) => (var_ok = {ref |-> TRUE})))))))))

INITIALISATION
    assert_ok := TRUE;
    is_initialisation := TRUE;
    var_ok := {};
    var_only_exec_up_or_down := {};
    var_not_lower_than_lowest_floor := {};
    var_not_higher_than_highest_floor := {}

OPERATIONS
    clock_step(param_up, param_down) =
        PRE
            ((param_up : LUSTRE_BOOL_NOT_NIL) & (param_down : LUSTRE_BOOL_NOT_NIL))
        THEN
            VAR var_up, var_down, var_exec_up, var_exec_down, var_floor IN
                var_up := param_up;
                var_down := param_down;
                Node_1.clock_step(var_up, var_down);
                var_exec_up := Node_1.var_exec_up;
                var_exec_down := Node_1.var_exec_down;
                var_floor := Node_1.var_floor;
                PRE (l_fby_boolean(is_initialisation |-> l_and(l_not(var_up) |->
                    l_not(var_down)) |->
                    l_not(l_and(var_up |-> var_down))) /= {}) THEN skip END;
                assert_ok := bool(((assert_ok = TRUE) &
                            (l_fby_boolean(is_initialisation |->
                            l_and(l_not(var_up) |-> l_not(var_down)) |->
                            l_not(l_and(var_up |-> var_down))) = {ref |-> TRUE})));
                var_only_exec_up_or_down := l_not(l_and(var_exec_up |-> var_exec_down));
                var_not_lower_than_lowest_floor := l_greater_equal(var_floor |->
                                                    {ref |-> 0});
                var_not_higher_than_highest_floor := l_less_equal(var_floor |->
                                                    {ref |-> 100});
                var_ok := l_and(l_and(var_only_exec_up_or_down |->
                        var_not_lower_than_lowest_floor) |->
                        var_not_higher_than_highest_floor);
                SELECT ((assert_ok = TRUE) & (Node_1.assert_ok = TRUE)) THEN skip END;
                is_initialisation := FALSE

            END

        END
END

/*
Node_1: Lift(up, down)
*/
```

Lift:

```
MACHINE M_Lift

SEES LibraryLustre

VARIABLES assert_ok , is_initialisation , var_exec_up , var_exec_down , var_floor

INVARIANT ((assert_ok : BOOL) & ((is_initialisation : BOOL) & ((var_floor : LUSTRE_INT) &
((var_exec_down : LUSTRE_BOOL) & ((var_exec_up : LUSTRE_BOOL) &
(((is_initialisation = FALSE) => (var_floor /= {})) &
(((is_initialisation = FALSE) => (var_exec_down /= {})) &
((is_initialisation = FALSE) => (var_exec_up /= {})))))))))

INITIALISATION
    assert_ok := TRUE;
    is_initialisation := TRUE;
    var_exec_up := {};
    var_exec_down := {};
    var_floor := {}

OPERATIONS

    clock_step(param_up, param_down) =
        PRE
            ((param_up : LUSTRE_BOOL_NOT_NIL) & (param_down : LUSTRE_BOOL_NOT_NIL))
        THEN
            VAR var_up, var_down, var_old_floor IN
                var_up := param_up;
                var_down := param_down;
                var_old_floor := l_fby_integer(is_initialisation |-> {ref |-> 0} |->
                              var_floor);
                var_exec_up := l_and(var_up |-> l_less(var_old_floor |-> {ref |-> 100}));
                var_exec_down := l_and(var_down |-> l_greater(var_old_floor |->
                              {ref |-> 0}));
                var_floor := l_ite_integer(var_exec_up |-> l_plus(var_old_floor |->
                              {ref |-> 1}) |-> l_ite_integer(var_exec_down |->
                              l_minus(var_old_floor |-> {ref |-> 1}) |-> var_old_floor));
                SELECT (assert_ok = TRUE) THEN skip END;
                is_initialisation := FALSE

            END

        END



END
```

Translation of SLOW_TIME_STABLE.lus to B with SLOW_TIME_STABLE as main node:

SLOW_TIME_STABLE:

```
MACHINE M_SLOW_TIME_STABLE

INCLUDES Node_1.M_SLOW_STABLE

SEES LibraryLustre

VARIABLES assert_ok, is_initialisation_second, is_initialisation, var_level, var_second

INVARIANT ((assert_ok : BOOL) & ((is_initialisation_second : BOOL) &
((is_initialisation : BOOL) & ((var_second : LUSTRE_BOOL) & ((var_level : LUSTRE_BOOL) &
((is_initialisation_second = FALSE) => (var_level /= {}))))))

INITIALISATION
    assert_ok := TRUE;
    is_initialisation_second := TRUE;
    is_initialisation := TRUE;
    var_level := {};
    var_second := {}


OPERATIONS

    clock_step(param_set, param_second, param_delay) =
        PRE
            ((param_set : LUSTRE_BOOL_NOT_NIL) & ((param_second : LUSTRE_BOOL_NOT_NIL) &
            (param_delay : LUSTRE_INT_NOT_NIL)))
        THEN
            VAR var_set, var_delay IN
                var_set := param_set;
                var_second := param_second;
                PRE (var_second /= {}) THEN skip END;
                var_delay := param_delay;
                IF (var_second = {ref |-> TRUE}) THEN
                    Node_1.clock_step(var_set, var_delay)
                END;
                IF (var_second = {ref |-> TRUE}) THEN
                    var_level := Node_1.var_level
                END;
                SELECT ((assert_ok = TRUE) & (Node_1.assert_ok = TRUE)) THEN skip END;
                IF (var_second = {ref |-> TRUE}) THEN
                    is_initialisation_second := FALSE
                END;
                is_initialisation := FALSE

            END

        END



END

/*
Node_1: SLOW_STABLE((set, delay when second))
*/
```

SLOW_STABLE:

```
MACHINE M_SLOW_STABLE

SEES LibraryLustre

VARIABLES assert_ok , is_initialisation , var_level , var_count

INVARIANT ((assert_ok : BOOL) & ((is_initialisation : BOOL) & ((var_level : LUSTRE_BOOL)
& (var_count : LUSTRE_INT))))

INITIALISATION
    assert_ok := TRUE;
    is_initialisation := TRUE;
    var_level := {};
    var_count := {}


OPERATIONS

    clock_step(param_set, param_delay) =
        PRE
            ((param_set : LUSTRE_BOOL) & (param_delay : LUSTRE_INT))
        THEN
            VAR var_set, var_delay IN
                var_set := param_set;
                var_delay := param_delay;
                var_count := l_ite_integer(var_set |-> var_delay |->
                            l_ite_integer(l_fby_boolean(is_initialisation |->
                            {ref |-> FALSE} |-> var_level) |-> l_minus(var_count
                            |-> {ref |-> 1}) |-> {ref |-> 0}));
                var_level := l_greater(var_count |-> {ref |-> 0});
                is_initialisation := FALSE

            END

        END



END
```

Translation of UMS.lus:

```
MACHINE M_UMS

SEES LibraryLustre

VARIABLES assert_ok , is_initialisation , var_grant_access , var_grant_exit ,
var_do_AB , var_do_BC

INVARIANT ((assert_ok : BOOL) & ((is_initialisation : BOOL) &
((var_do_BC : LUSTRE_BOOL) & ((var_do_AB : LUSTRE_BOOL) &
((var_grant_exit : LUSTRE_BOOL) & ((var_grant_access : LUSTRE_BOOL) &
(((is_initialisation = FALSE) => (var_do_BC /= {})) &
(((is_initialisation = FALSE) => (var_do_AB /= {})) &
(((is_initialisation = FALSE) => (var_grant_exit /= {})) &
((is_initialisation = FALSE) => (var_grant_access /= {}))))))))))

INITIALISATION
    assert_ok := TRUE;
    is_initialisation := TRUE;
    var_grant_access := {};
    var_grant_exit := {};
    var_do_AB := {};
    var_do_BC := {}


OPERATIONS

    clock_step(param_on_A, param_on_B, param_on_C, param_ack_AB, param_ack_BC) =
        PRE
            ((param_on_A : LUSTRE_BOOL_NOT_NIL) & ((param_on_B : LUSTRE_BOOL_NOT_NIL) &
            ((param_on_C : LUSTRE_BOOL_NOT_NIL) & ((param_ack_AB : LUSTRE_BOOL_NOT_NIL) &
            (param_ack_BC : LUSTRE_BOOL_NOT_NIL)))))
        THEN
            VAR var_on_A, var_on_B, var_on_C, var_ack_AB, var_ack_BC, var_empty_section,
              var_only_on_B IN
                var_on_A := param_on_A;
                var_on_B := param_on_B;
                var_on_C := param_on_C;
                var_ack_AB := param_ack_AB;
                var_ack_BC := param_ack_BC;
                var_empty_section := l_not(l_or(l_or(var_on_A |-> var_on_B) |->
                                var_on_C));
                var_only_on_B := l_and(var_on_B |-> l_not(l_or(var_on_A |-> var_on_C)));
                var_grant_access := l_and(var_empty_section |-> var_ack_AB);
                var_grant_exit := l_and(var_only_on_B |-> var_ack_BC);
                var_do_AB := l_and(l_not(var_ack_AB) |-> var_empty_section);
                var_do_BC := l_and(l_not(var_ack_BC) |-> var_only_on_B);
                SELECT (assert_ok = TRUE) THEN skip END;
                is_initialisation := FALSE

            END

        END



END
```

F Microbenchmarks Results

F.1 Logical Operations

Table 27: Runtimes of Microbenchmarks for Logical Operations According to LUSTRE AST interpretation, Translation to B, and Code Generation to Java and C++ in Seconds, Memory Usage in KB, PI = Primitive Integer

and		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	27.31	5.71	2.70	2.10	2.08
	Memory	1 343 384	167 168	200 652	776	780
or		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	28.91	5.04	2.64	1.58	1.62
	Memory	1 343 360	167 208	209 488	792	796
not		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	23.98	4.09	2.32	1.10	1.15
	Memory	1 343 348	167 192	165 684	776	780

F.2 Arithmetic Operations

Table 28: Runtimes of Microbenchmarks for Arithmetic Operations According to LUSTRE AST interpretation, Translation to B, and Code Generation to Java and C++ in Seconds, Memory Usage in KB, PI = Primitive Integer

plus (integer)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	90.77	4.63	2.80	1.55	1.58
	Memory	2 103 120	167 224	205 708	796	800
plus (real)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	Not Supported	4.79	Not Supported	Not Supported	Not Supported
	Memory	Not Supported	167 196	Not Supported	Not Supported	Not Supported
minus (integer)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	93.98	4.60	2.54	1.52	1.60
	Memory	2 103 084	167 208	209 832	812	816
minus (real)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	Not Supported	4.62	Not Supported	Not Supported	Not Supported
	Memory	Not Supported	167 208	Not Supported	Not Supported	Not Supported
unary minus (integer)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	59.38	4.02	2.42	1.20	1.11
	Memory	2 103 108	167 204	172 108	796	800
unary minus (real)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	Not Supported	4.05	Not Supported	Not Supported	Not Supported
	Memory	Not Supported	167 176	Not Supported	Not Supported	Not Supported
multiply (integer)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	90.18	4.59	2.74	1.58	1.51
	Memory	2 103 112	167 172	205 340	812	816
multiply (real)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	Not Supported	4.65	Not Supported	Not Supported	Not Supported
	Memory	Not Supported	167 208	Not Supported	Not Supported	Not Supported
divide (integer)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	113.96	4.66	3.02	2.11	1.99
	Memory	2 103 120	167 208	210 664	812	816
divide (real)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	Not Supported	4.62	Not Supported	Not Supported	Not Supported
	Memory	Not Supported	167 244	Not Supported	Not Supported	Not Supported
modulo		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	124.80	4.53	3.08	2.06	1.99
	Memory	2 103 136	167 188	205 508	812	812

F.3 Comparisons

Table 29: Runtimes of Microbenchmarks for Comparisons According to LUSTRE AST interpretation, Translation to B, and Code Generation to Java and C++ in Seconds, Memory Usage in KB, PI = Primitive Integer

less (integer)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	102.96	5.08	2.83	1.53	1.59
	Memory	1 343 448	167 172	201 652	812	820
less (real)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	Not Supported	4.62	Not Supported	Not Supported	Not Supported
	Memory	Not Supported	167 180	Not Supported	Not Supported	Not Supported
less_equal (integer)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	105.12	4.77	2.81	1.60	1.57
	Memory	1 343 404	167 208	209 180	812	820
less _equal (real)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	Not Supported	4.73	Not Supported	Not Supported	Not Supported
	Memory	Not Supported	167 180	Not Supported	Not Supported	Not Supported
greater (integer)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	108.98	4.57	2.99	1.59	1.65
	Memory	1 343 416	167 196	200 180	812	820
greater (real)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	Not Supported	4.86	Not Supported	Not Supported	Not Supported
	Memory	Not Supported	167 216	Not Supported	Not Supported	Not Supported
greater_equal (integer)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	97.02	4.86	2.97	1.56	1.59
	Memory	1 343 440	167 176	204 404	796	804
greater_equal (real)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	Not Supported	4.72	Not Supported	Not Supported	Not Supported
	Memory	Not Supported	167 184	Not Supported	Not Supported	Not Supported
equal (integer)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	81.78	4.68	2.78	1.52	1.61
	Memory	2 103 104	167 168	200 436	796	804
equal (real)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	Not Supported	4.57	Not Supported	Not Supported	Not Supported
	Memory	Not Supported	167 224	Not Supported	Not Supported	Not Supported
equal (boolean)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	31.01	4.67	2.91	1.55	1.56
	Memory	1 343 364	167 208	201 292	776	780
unequal (integer)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	77.48	4.71	3.48	1.60	1.59
	Memory	1 343 452	167 212	203 080	812	820
unequal (real)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	Not Supported	4.77	Not Supported	Not Supported	Not Supported
	Memory	Not Supported	167 204	Not Supported	Not Supported	Not Supported
unequal (boolean)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	33.54	4.78	3.88	1.55	1.58
	Memory	1 343 364	167 208	201 412	792	796

F.4 If-Then-Else

Table 30: Runtimes of Microbenchmarks for If-Then-Else According to LUSTRE AST interpretation, Translation to B, and Code Generation to Java and C++ in Seconds, Memory Usage in KB, PI = Primitive Integer

if-then-else (integer)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	104.72	5.09	3.59	1.99	2.08
	Memory	1 343 460	167 220	205 212	800	808
if-then-else (real)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	Not Supported	5.14	Not Supported	Not Supported	Not Supported
	Memory	Not Supported	167 228	Not Supported	Not Supported	Not Supported
if-then-else (boolean)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	28.57	5.04	3.62	1.96	1.98
	Memory	1 343 364	167 180	203 824	792	796

F.5 Temporal Operations

Table 31: Runtimes of Microbenchmarks for Temporal Operations as Interpreted LUSTRE Programs, Translated B Models and Generated Code in Seconds, Memory Usage in KB, PI = Primitive Integer

current (integer)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	48.62	5.36	4.18	1.84	1.86
	Memory	2 103 132	167 180	204 124	796	800
current (real)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	Not Supported	5.35	Not Supported	Not Supported	Not Supported
	Memory	Not Supported	167 196	Not Supported	Not Supported	Not Supported
current (boolean)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	37.02	5.64	3.94	1.94	1.79
	Memory	2 103 020	167 168	205 376	776	780
when (integer)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	40.13	4.77	3.85	1.81	1.81
	Memory	2 103 100	167 172	204 180	800	808
when (real)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	Not Supported	4.71	Not Supported	Not Supported	Not Supported
	Memory	Not Supported	165 068	Not Supported	Not Supported	Not Supported
when (boolean)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	31.16	4.67	3.71	1.82	1.78
	Memory	1 343 400	167 212	202 388	792	796
-> (integer)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	77.52	4.66	3.29	1.63	1.64
	Memory	2 103 076	167 220	201 504	796	800
-> (real)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	Not Supported	4.89	Not Supported	Not Supported	Not Supported
	Memory	Not Supported	167 168	Not Supported	Not Supported	Not Supported
-> (boolean)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	25.24	4.60	3.42	1.61	1.55
	Memory	1 343 360	167 204	202 244	792	800
pre (integer)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	29.52	4.18	2.52	0.79	0.64
	Memory	1 343 408	167 208	146 400	792	796
pre (real)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	Not Supported	4.17	Not Supported	Not Supported	Not Supported
	Memory	Not Supported	167 208	Not Supported	Not Supported	Not Supported
pre (boolean)		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	20.60	4.08	2.87	0.73	0.74
	Memory	1 343 360	167 172	153 496	776	780

F.6 Other

Table 32: Runtimes of Microbenchmarks for Assertions and Many Clocks According to LUSTRE AST interpretation, Translation to B, and Code Generation to Java and C++ in Seconds, Memory Usage in KB, PI = Primitive Integer

Assertion		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	38.42	5.40	3.08	1.99	2.06
	Memory	2 103 064	167 168	203 556	792	804
Node Call		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	39.99	7.65	2.57	1.63	1.64
	Memory	1 343 404	167 184	201 572	800	808
Many Clocks		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	332.21	16.80	20.92	23.69	23.79
	Memory	3 457 120	167 204	273 060	804	812
Many Local Clocks		B	Lustre	Java PI	C++ PI -O1	C++ PI -O2
	Runtime	224.17	18.30	19.62	21.34	21.48
	Memory	3 334 076	167 220	318 720	800	812

G Model Checking Appendix

Pilot Flying Properties

See `https://gitlab.cs.uni-duesseldorf.de/vu/lustre-interpreter` for more details.

Table 33: Runtimes of BMC(50) Applied on Decomposed Properties in Pilot Flying with True Properties, Memory Usage in KB, TO = Timeout

Property 1		BMC(50)
	Runtime	TO at Level 5
	Memory	170 640
Property 2		BMC(50)
	Runtime	TO at Level 4
	Memory	170 660
Property 3		BMC(50)
	Runtime	TO at Level 5
	Memory	170 672
Property 4		BMC(50)
	Runtime	TO at Level 7
	Memory	170 668
Property 5		BMC(50)
	Runtime	TO at Level 4
	Memory	170 648
Property 6		BMC(50)
	Runtime	TO at Level 7
	Memory	170 680
Property 7		BMC(50)
	Runtime	TO at Level 7
	Memory	170 620
Property 8		BMC(50)
	Runtime	TO at Level 7
	Memory	170 632
Property 9		BMC(50)
	Runtime	TO at Level 7
	Memory	170 640
Property 10		BMC(50)
	Runtime	TO at Level 7
	Memory	170 640
Property 11		BMC(50)
	Runtime	TO at Level 7
	Memory	170 676
Property 12		BMC(50)
	Runtime	TO at Level 7
	Memory	170 568

Table 34: Runtimes of BMC(50) Applied on Decomposed Properties in Pilot Flying with True Properties, Memory Usage in KB, TO = Timeout

Property 13		BMC(50)
	Runtime	TO at Level 7
	Memory	169 348
Property 14		BMC(50)
	Runtime	TO at Level 7
	Memory	167 708
Property 15		BMC(50)
	Runtime	TO at Level 7
	Memory	167 264
Property 16		BMC(50)
	Runtime	TO at Level 7
	Memory	167 308
Property 17		BMC(50)
	Runtime	TO at Level 7
	Memory	167 268
Property 18		BMC(50)
	Runtime	TO at Level 7
	Memory	167 260
Property 19		BMC(50)
	Runtime	TO at Level 7
	Memory	167 284
Property 20		BMC(50)
	Runtime	TO at Level 7
	Memory	167 268
Property 21		BMC(50)
	Runtime	TO at Level 7
	Memory	167 284
Property 22		BMC(50)
	Runtime	TO at Level 7
	Memory	170 688
Property 23		BMC(50)
	Runtime	TO at Level 7
	Memory	170 680
Property 24		BMC(50)
	Runtime	TO at Level 7
	Memory	170 688

Table 35: Runtimes of Explicit-State Model Checking and BMC for Pilot Flying with False Properties in Seconds with Number of States and Transitions, Memory Usage in KB, TO = Timeout, EF = Error Found, BF = Breadth-First Serach, DF = Depth-First Search

Property 4 Error		B BF	B DF	Lustre BF	Lustre DF	BMC(50)
	Runtime	0.31 (EF)	0.35 (EF)	0.08 (EF)	0.08 (EF)	0.42 at Level 1 (EF)
	Memory	171 128	171 128	167 532	167 524	170 832
	States	1	1	1	1	-
	Transitions	62	62	61	61	-
	States/s	3	3	12	12	-
	Transitions/s	200	177	762	762	-
Property 8 Error		B BF	B DF	Lustre BF	Lustre DF	BMC(50)
	Runtime	0.31 (EF)	0.35 (EF)	0.08 (EF)	0.08 (EF)	0.43 at Level 1 (EF)
	Memory	171 152	171 128	167 540	167 544	170 812
	States	1	1	1	1	-
	Transitions	62	62	61	61	-
	States/s	3	3	12	12	-
	Transitions/s	200	177	762	762	-
Property 15 Error		B BF	B DF	Lustre BF	Lustre DF	BMC(50)
	Runtime	0.31 (EF)	0.35 (EF)	0.09 (EF)	0.09 (EF)	0.44 at Level 1 (EF)
	Memory	171 152	171 156	167 552	167 488	170 824
	States	1	1	1	1	-
	Transitions	62	62	61	61	-
	States/s	3	3	11	11	-
	Transitions/s	200	177	677	677	-
Property 19 Error		B BF	B DF	Lustre BF	Lustre DF	BMC(50)
	Runtime	TO after 20 min	TO after 20 min	620.87 (EF) [18]	TO after 20 min	TO at Level 4
	Memory	382 940	762 220	401 652	739 680	170 760
	States	14 576	7701	20 124	43 500	-
	Transitions	176 246	230 972	240 143	335 611	-
	States/s	12	6	32	36	-
	Transitions/s	146	192	386	279	-

[18] Counterexample with length 6 including initializing XTL system

Submode Logic Properties

See `https://gitlab.cs.uni-duesseldorf.de/vu/lustre-interpreter` for more details.

Table 36: Runtimes of BMC(50) Applied on Decomposed Properties in Submode Logic with True Properties, Memory Usage in KB, TO = Timeout

Property 1		BMC(50)
	Runtime	TO at Level 8
	Memory	169 876
Property 2		BMC(50)
	Runtime	TO at Level 8
	Memory	169 848
Property 3		BMC(50)
	Runtime	TO at Level 8
	Memory	169 888
Property 4		BMC(50)
	Runtime	TO at Level 8
	Memory	169 872

Table 37: Runtimes of Explicit-State Model Checking and BMC for Submode Logic with False Properties in Seconds with Number of States and Transitions, Memory Usage in KB, TO = Timeout, EF = Error Found, BF = Breadth-First Serach, DF = Depth-First Search

Property 1 Error		B BF	B DF	Lustre BF	Lustre DF	BMC(50)
	Runtime	0.06 (EF)	0.06 (EF)	0.03 (EF)	0.04 (EF)	0.35 at Level 1 (EF)
	Memory	170 180	170 156	167 484	167 484	169 988
	States	1	1	1	1	-
	Transitions	18	18	17	17	-
	States/s	16	16	33	25	-
	Transitions/s	300	300	566	425	-
Property 4 Error		B BF	B DF	Lustre BF	Lustre DF	BMC(50)
	Runtime	0.06 (EF)	0.06 (EF)	0.03 (EF)	0.04 (EF)	0.46 at Level 1 (EF)
	Memory	170 172	170 180	167 488	167 488	170 008
	States	1	1	1	1	-
	Transitions	18	18	17	17	-
	States/s	16	16	33	25	-
	Transitions/s	300	300	566	425	-

Docking Example Properties

See `https://gitlab.cs.uni-duesseldorf.de/vu/lustre-interpreter` for more details.

Table 38: Runtimes of BMC(50) Applied on Decomposed Properties in Docking Approach with True Properties, Memory Usage in KB, TO = Timeout

Property 1		BMC(50)
	Runtime	TO at Level 4
	Memory	235 140
Property 2		BMC(50)
	Runtime	TO at Level 5
	Memory	290 456
Property 3		BMC(50)
	Runtime	TO at Level 3
	Memory	209 264
Property 4		BMC(50)
	Runtime	TO at Level 3
	Memory	202 332
Property 5		BMC(50)
	Runtime	TO at Level 4
	Memory	211 784
Property 6		BMC(50)
	Runtime	TO at Level 5
	Memory	280 160
Property 7		BMC(50)
	Runtime	TO at Level 10
	Memory	541 480
Property 8		BMC(50)
	Runtime	TO at Level 4
	Memory	188 440
Property 9		BMC(50)
	Runtime	TO at Level 4
	Memory	189 236
Property 10		BMC(50)
	Runtime	TO at Level 3
	Memory	181 224
Property 11		BMC(50)
	Runtime	TO at Level 4
	Memory	187 920

Note 1: The implementation of the Docking Approach with False Properties has slight differences compared to the implementation with True Properties and the Docking Approach used in Section 8. Note 2: Property 4 is true and is thus not part of the table

See `https://gitlab.cs.uni-duesseldorf.de/vu/lustre-interpreter` for more details.

Table 39: Runtimes of BMC(50) Applied on Decomposed Properties in Docking Approach with False Properties, Memory Usage in KB, TO = Timeout, EF = Error Found

Property 1 Error		BMC(50)
	Runtime	TO at Level 5
	Memory	421 560
Property 2 Error		BMC(50)
	Runtime	TO at Level 5
	Memory	380 184
Property 3 Error		BMC(50)
	Runtime	TO at Level 5
	Memory	381 484
Property 5 Error		BMC(50)
	Runtime	TO at Level 4
	Memory	318 888
Property 6 Error		BMC(50)
	Runtime	TO at Level 4
	Memory	352 856
Property 7 Error		BMC(50)
	Runtime	TO at Level 4
	Memory	347 104
Property 8 Error		BMC(50)
	Runtime	TO at Level 4
	Memory	334 340
Property 9 Error		BMC(50)
	Runtime	TO at Level 4
	Memory	337 052
Property 10 Error		BMC(50)
	Runtime	TO at Level 4
	Memory	375 320
Property 11 Error		BMC(50)
	Runtime	TO at Level 4
	Memory	381 548

Note: This program is Docking Approach with fewer input parameters. Despite `StageTransition`, all integer parameters in the original version are set to 2 and defined as local variables. Furthermore, the safety property is defined as `not(JointMission)`. This makes it possible to find a trace where the Docking Approach is completed. Furthermore, this is the only Model Checking benchmark where the timeout is not equal 20 minutes. It is set to one hour for demonstration that such a path can be find with model checking.

See `https://gitlab.cs.uni-duesseldorf.de/vu/lustre-interpreter` for more details.

Table 40: Runtimes of Explicit-State Model Checking and BMC for Docking Approach GOAL in Seconds with Number of States and Transitions, Memory Usage in KB, TO = Timeout, EF = Error Found, BF = Breadth-First Serach, DF = Depth-First Search

Docking Approach GOAL		B BF	B DF	Lustre BF	Lustre DF	BMC(50)
	Runtime	TO at Start	TO at Start	2978.86 (EF[19])	2446.03 (EF[20])	TO at Level 4
	Memory	177 448	177 420	784 068	706 980	235 692
	States	-	-	7088	6415	-
	Transitions	-	-	1 361 089	1 231 873	-
	States/s	-	-	2	2	-
	Transitions/s	-	-	456	503	-

[19]Counterexample with length 27 including initializing XTL system

[20]Counterexample with length 1100 including initializing XTL system

References

[1] J.R. Abrial and Hoare, A. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 2005.

[2] Daniel Dollé and Didier Essamé and Jérôme Falampin. B dans le transport ferroviaire. L'expérience de Siemens Transportation Systems. *Technique et Science Informatiques*, 22(1):11–32, 2003.

[3] Dominik Hansen and Michael Leuschel and David Schneider and Sebastian Krings and Philipp Körner and Thomas Naulin and Nader Nayeri and Frank Skowron. Using a Formal B Model at Runtime in a Demonstration of the ETCS Hybrid Level 3 Concept with Real Trains. In Michael Butler and Alexander Raschke and Thai Son Hoang and Klaus Reichl, editor, *Proceedings ABZ 2018*, volume 10817 of *LNCS*, pages 292–306. Springer-Verlag.

[4] Michael Leuschel and Michael Butler. ProB: A Model Checker for B. In Araki Keijiro, Stefania Gnesi, and Mandrio Dino, editors, *FME*, volume 2805 of *Lecture Notes in Computer Science*, pages 855–874. Springer-Verlag, 2003.

[5] Plagge, Daniel and Leuschel, Michael. Seven at a stroke: LTL model checking for high-level specifications in B, Z, CSP, and more. *International Journal on Software Tools for Technology Transfer*, 12:9–21, 01 2007.

[6] Krings, Sebastian. *Towards Infinite-State Symbolic Model Checking for B and Event-B*. PhD thesis, Heinrich Heine Universität Düsseldorf, August 2017.

[7] Vu, Fabian and Hansen, Dominik and Körner, Philipp and Leuschel, Michael. A Multi-target Code Generator for High-Level B. In Ahrendt, Wolfgang and Tapia Tarifa, Silvia Lizeth, editor, *Integrated Formal Methods*, pages 456–473, Cham, 2019. Springer International Publishing.

[8] Nicolas Halbwachs. A Synchronous Language at Work: The Story of Lustre. volume 2005, pages 3– 11, 08 2005.

[9] Brat, Guillaume and Bushnell, David and Davies, Misty and Giannakopoulou, Dimitra and Howar, Falk and Kahsai, Temesghen. Verifying the Safety of a Flight-Critical System. In Bjørner, Nikolaj and de Boer, Frank, editor, *FM 2015: Formal Methods*, pages 308–324, Cham, 2015. Springer International Publishing.

[10] Kahsai, Temesghen and Garoche, Pierre-Loïc and Tinelli, Cesare and Whalen, Mike. Incremental Verification with Mode Variable Invariants in State Machines. In Goodloe, Alwyn E. and Person, Suzette, editor, *NASA Formal Methods*, pages 388–402, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[11] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, Sep. 1991.

[12] Halbwachs, Nicolas. Synchronous Programming of Reactive Systems. 1427, 12 1999.

[13] Erwan Jahier. Verimag Development Tools for Critical Reactive Systems using the Synchronous Approach The Lustre-V6 Tool Box Website. `http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v6/`, 2019. Accessed: 2019-10-15.

[14] P. Caspi and D. Pilaud and N. Halbwachs and J. A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *14th Symposium on Principles of Programming Languages (POPL'87). ACM*, 1987.

[15] Nicolas Halbwachs Erwan Jahier, Pascal Raymond. The Lustre V6 Reference Manual. `http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v6/doc/lv6-ref-man.pdf`. Accessed: 2019-10-01.

[16] François Xavier Dormoy. Scade 6 a model based solution for safety critical software development. 2007.

[17] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language lustre. *Software Engineering, IEEE Transactions on*, 18:785 – 793, 10 1992.

[18] Verimag Imag. Formal Verification, Theory, Techniques and Tools. `https://www-verimag.imag.fr/Formal-Verification-Theory.html`. Accessed: 2020-03-18.

[19] ClearSy. *Atelier B, User and Reference Manuals*. Aix-en-Provence, France, 2016. Available at `http://www.atelierb.eu/`.

[20] Gagnon, Etienne and Hendren, Laurie. SableCC An Object-Oriented Compiler Framework. *Proceedings of TOOLS 1998*, 04 1998.

[21] Terence John Parr and Russell W. Quong. ANTLR: A Predicated- LL(k) Parser Generator. *Softw., Pract. Exper.*, 25:789–810, 1995.

[22] Bundy, Alan and Wallen, Lincoln. *Definite Clause Grammars*, page 26. Springer Berlin Heidelberg, Berlin, Heidelberg, 1984.

[23] Erich Gamma and Richard Helm. Visitor. In *Design Patterns: Elements of Reusable Object-Oriented Software*, pages 331–344. Addison Wesley, 1994.

[24] G. M. Skii and Ye. M. Landis. An algorithm for the organization of information. 1962.

[25] J.-L Bergerand, P Caspi, N Halbwachs, and J Plaice. Automatic control systems programming using a real-time declarative language. 12 1987.

[26] John Plaice. Nested clocks: The lustre synchronous dataflow language. 1988.

[27] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[28] N. Halbwachs. A Tutorial of Lustre, 1993.

[29] *Formal Development of Reactive Systems - Case Study Production Cell*, Berlin, Heidelberg, 1995. Springer-Verlag.

[30] Erwan Jahier. Lustre Examples Github Repository Website. `https://github.com/jahierwan/lustre-examples`, 2017. Accessed: 2020-06-09.

[31] Temesghen Kahsai. BDD based invariant generation for Lustre programs BitBucket Repository Website. `https://bitbucket.org/lememta/bdd-inv`, 2014. Accessed: 2020-06-09.

[32] Margarita Sampson and Vladimir Derevenko. Interface definition document (idd) for international space station (iss) visiting vehicles (vvs). *NASA Technical Report*, 2000.

[33] Oracle. The Java HotSpot Performance Engine Architecture. `https://www.oracle.com/technetwork/java/whitepaper-135217.html`. Accessed: 2020-05-27.

[34] Raymond, Pascal. *Synchronous Program Verification with Lustre/Lesar*, pages 171 – 206. 01 2010.

[35] J.A. Plaice and J.-B. Saint. *The LUSTRE-ESTEREL portable format*. INRIA, CMA, Sophia Antipolis, France, September 1987. unpublished report.

[36] Bourke, Timothy and Brun, Lélio and Dagand, Pierre-Évariste and Leroy, Xavier and Pouzet, Marc and Rieg, Lionel. A Formally Verified Compiler for Lustre. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 586–601, New York, NY, USA, 2017. Association for Computing Machinery.

[37] Bourke, Timothy and Colaço, Jean-Louis and Pagano, Bruno and Pasteur, Cédric and Pouzet, Marc. A Synchronous-Based Code Generator for Explicit Hybrid Systems Languages. In Franke, Björn, editor, *Compiler Construction*, pages 69–88, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[38] Tripakis, Stavros and Sofronis, Christos and Caspi, Paul and Curic, Adrian. Translating Discrete-Time Simulink to Lustre. *ACM Trans. Embed. Comput. Syst.*, 4(4):779–818, November 2005.

[39] Scaife, N. and Sofronis, C. and Caspi, P. and Tripakis, S. and Maraninchi, F. Defining and Translating a "Safe" Subset of Simulink/Stateflow into Lustre. In *Proceedings of the 4th ACM International Conference on Embedded Software*, EMSOFT '04, page 259–268, New York, NY, USA, 2004. Association for Computing Machinery.

[40] Caspi, Paul and Curic, Adrian and Maignan, Aude and Sofronis, Christos and Tripakis, Stavros and Niebert, Peter. From Simulink to SCADE/Lustre to TTA: A Layered Approach for Distributed Embedded Applications. *SIGPLAN Not.*, 38(7):153–162, June 2003.

[41] Hamza Bourbouh, Pierre-Loïc Garoche, Thomas Loquen, Eric Noulard, and Claire Pagetti. Cocosim, a code generation framework for control/command applications: An overview of cocosim for multi-periodic discrete simulink models. 2020.

[42] Temesghen Kahsai and Cesare Tinelli. PKind: A parallel k-induction based model checker. *Electronic Proceedings in Theoretical Computer Science*, 72:55–62, Oct 2011.

[43] Andrew Gacek and John Backes and Mike Whalen and Lucas G. Wagner and Elaheh Ghassabani. The JKind Model Checker. *CoRR*, abs/1712.01222, 2017.

[44] Jeannet, Bertrand and Halbwachs, Nicolas and Raymond, Pascal. Dynamic Partitioning in Analyses of Numerical Properties. In Agostino Cortesi and Gilberto Filé, editors, *Static Analysis*, pages 39–50, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[45] K. Altisen and M. Moy. ac2lus: Bringing SMT-Solving and Abstract Interpretation Techniques to Real-Time Calculus through the Synchronous Language Lustre. In *2010 22nd Euromicro Conference on Real-Time Systems*, pages 207–216, July 2010.

[46] Caspi, Paul. A PVS Proof Obligation Generator for Lustre Programs. 02 2001.

[47] A. Mebsout and C. Tinelli. Proof certificates for SMT-based model checkers for infinite-state systems. In *2016 Formal Methods in Computer-Aided Design (FMCAD)*, pages 117–124, Oct 2016.

List of Figures

List of Tables

Listings

List of Algorithms