DOI:10.1145/3747584

To view the accompanying Technical Perspective, visit doi.acm.org/10.1145/3768311

Equation-Directed Axiomatization of Lustre Semantics to Enable Optimized Code Validation

By Christophe Garion, Lélio Brun, Pierre-Loïc Garoche, and Xavier Thirioux

ABSTRACT

Model-based design tools are often used to design safety-critical embedded software. Consequently, generating correct code from such models is crucial. We tackle this challenge on Lustre, a dataflow synchronous language that embodies the concepts that base such tools. Instead of proving correct a whole code generator, we turn an existing compiler into a certifying compiler from Lustre to C, following a translation validation approach.

We propose a solution that generates both C code and an attached specification expressing a correctness result for the generated and optionally optimized code. The specification yields proof obligations that are discharged by external solvers through the Frama-C platform.

1. INTRODUCTION

Model-based design tools, such as SCADE Suite or Simulink, are widely used to design control software. They provide engineers with an interface to build high-level applications based on block diagrams and state machines, and with code generators that translate these models into sequential code. These tools are based on synchronous dataflow languages such as Lustre,[9] which provides specific constructs to compose functions over infinite streams of values, making it well suited for designing control software targeting embedded systems. It is used as a kernel language for SCADE Suite[14] and can encode a subset of the discrete part of Simulink.[6]

Languages of the dataflow synchronous family usually share well-studied formal semantics and compilation techniques, allowing traceability, industrial certification, and verification. In the domain of safety-critical embedded software design, these features are paramount to ensuring strong guarantees on the generated executable code. In particular, the existence of a well-founded mathematical model to express the semantics of these languages makes them intrinsically suitable to the application of formal methods. While recent work formalizes the semantics of a Lustre subset in a prototype compiler[7] whose correctness is verified once and for all in the Coq proof assistant, we choose another approach to verified compilation: translation validation.[19] In this approach, the preservation of the semantics between the source program and the compiled one is checked for each run, *after* the compilation. In this paper, we show how we modify the existing Lustre-to-C compiler, LustreC, into a generator of both executable code and associated specification. This specification encodes a complete state/transition semantics of the source Lustre code and states that the generated code complies with this semantics, asserting the correctness of the generation process. The specification is yet abstract enough to support different levels of code optimizations. As an application, we target the Frama-C platform[2] and its specification language ACSL. Frama-C allows interfacing with external SMT solvers to check that the generated C code complies with its specification. Both the generated C code and its specification as pre/post function contracts follow the node modular approach,[4] which prevails in modern Lustre code generators such as SCADE Suite. While some Lustre model-checking tools[11] provide a node-modular axiomatization of Lustre semantics, the produced predicates, typically built as a large conjunction of flow equations semantics formulas allowing to check the correctness of the corresponding Lustre program, are usually difficult to prove. In this paper, we propose a logical encoding that relies on *composition* rather than *conjunction*. This approach, while semantically equivalent, is shown to be compatible with proof at code level. Our approach spares the burden of proving correct a whole feature-rich compiler in an interactive proof assistant by delegating the proof effort.

To summarize, with respect to the state of the art, our contribution is: a node-modular, equation-driven, axiomatization of Lustre semantics that is associated to each generated instruction—to enable automatic validation—and is compatible with several optimizations at code level.

The paper is organized as follows: Section 2 presents an overview of related works. Section 3 describes the syntax, semantics, and compilation process of the Lustre input language. Section 4 explains how we axiomatize Lustre semantics as a composition of equation-specific predicates and define a certifying compiler by adding specification to the generated code. Section 5 details optimization of generated code and associated annotations. We present some experimental results in section 6 and give concluding remarks and perspectives in section 7.

2. RELATED WORK

There have been endeavors for building verified compilers

The original version of this paper was published in *ACM Trans. on Embedded Computing Systems 22*, 5s (2023), 1–24.

for synchronous languages. The goal of the GeneAuto project[24] was to develop a qualified code generator for a subset of Simulink, with parts proved in Coq. Some preliminary work[25] showed semantics preservation results for some passes of a compiler for the Signal language.[3] To our knowledge, more advanced solutions focus on Lustre[23] (give an end-to-end correctness proof from an imperatively defined dataflow semantics to the semantics of C), while the Vélus compiler[7] uses a stream-based dataflow semantics and is built on the verified C compiler CompCert.[18] These solutions are proofs-of-concept prototypes that treat a restricted subset of the input languages. Our aim is different, since we want to extend a feature-rich existing compiler with certifying abilities. The main advantage is to sidestep the burden of having to reprove systematically the compiler when a variation is made in the compilation process. Indeed, LustreC is rather large software with about 40,000 lines of OCaml code, as it is designed as an experimental playground for Lustre compilation, with several additional features. Vélus is equally large with about 40,000 lines of Coq code, but the extracted code used to build the actual compiler only amounts to about 1,500 lines of OCaml code. This comparison highlights the fact that the two approaches actually aim for different goals. Vélus is an experimental proof that shows it is possible to prove the correctness of the compilation of Lustre in its simplest form. As the main effort is on the formalization and proofs, the compilation scheme is designed with the correctness proof in mind and is kept as simple as possible. Our work seeks to demonstrate using translation-validation techniques to verify the correctness of an existing feature-rich compiler, without impacting the compilation scheme in itself, which can remain arbitrarily complex. In this paper, we nonetheless focus on a subset close to the one treated by Vélus to assess the feasibility of our approach. The level of insurance in the generated code verified using translation validation techniques or a verified compiler is the same *if* the validation process, that is, the *validator*, is itself formally verified. Notice it is not strictly the case in this work: The trust is deferred onto the SMT solvers. While a reasonable level of trust can be placed in them, these solvers are not formally proved correct.

Translation validation[19] is an approach that was early applied to synchronous languages.[20] Following this approach, the semantics preservation is not proved once and for all by proving a compiler, but verified *a posteriori* for each run of the compiler. Research in this domain about synchronous languages concentrates mainly on Signal[1,12,21] and Simulink.[10,22] In particular, Cavalcanti et al.[10] proposes a framework to show refinement relations between Simulink discrete-time block diagrams and SPARK/Ada implementations. These works and our solution, which specifically targets compilation from Lustre to C, are in the same vein. Finkbeiner et al.[16] follow essentially the same approach as ours: From monitors written in the Lola stream-based synchronous specification language, they generate Rust code annotated with specification targeting the verification platform Viper. The authors mainly focus on minimizing the memory footprint of generated monitors. Both Calvalcanti et al.[10] and Finkbeiner et al.[16] handle a rather simple input language, lacking advanced control structures such as clock sampling, resetting, and state machines. Furthermore, it seems the proposed approaches have been tested against a limited set of modest examples. In contrast, we use modern Lustre as input, with all the aforementioned features. As we also emphasize scalability, we applied our method to hundreds of use cases, including real-life industrial examples.

While we restrict our approach to discrete-time synchronous systems, there exist proposals combining several approaches to specifically tackle design and verification of hybrid systems. The MARS[13] framework provides an integrated solution to design and verify hybrid Simulink models. Several rewriting steps are used, and verification is performed by simulation. VeriPhy[5] is a toolchain focusing on hybrid cyber-physical systems, built around several provers, that provides a proof that properties are preserved from high-level models to controller executables. VeriPhy is closer to verified compilers: a chain of rewriting steps that are individually proven correct in different provers.

3. THE LUSTRE LANGUAGE

We present the Lustre language with a simple counter modulo 4 example. The Lustre code is presented in Figure 1a. We define a *node* called `count` that is a stream function without input that outputs a boolean stream out. The output and local streams are defined by equations whose order is insignificant. The local stream `time` and the output stream `out` are defined using simple equations: Literal constants represent constant streams, arithmetic operators operate point-wise, and `if/then/else` is a multiplexer. The stream `time` is also defined with the `->` operator: It has the value `0` at the initial instant and the value of the righthand-side expression otherwise. The `pre` operator represents an uninitialized delay.

A dataflow representation of the execution is shown in Figure 2. Each variable or expression is associated with its corresponding stream. The columns give the values of the streams indexed at each successive instant. We can clearly describe the behavior of the `pre` operator: The stream associated with `pre time` is the stream associated with `time` delayed by one instant, where $\perp$ represents the uninitialized value. On the right is represented a state/transition system execution. Under this view, the node is considered as a system with an internal state, whose evolution is dictated by transitions. Successive transitions, labeled with the indexed output values, encode the node equations. The state is a tree, where nodes are Lustre node sub-instances and leaves are bindings between state variables and their values.

3.1. Compiler architecture.

The standard Lustre compilation approach, described in Biernacki et al.,[4] consists of a single-loop modular scheme, where a sequential *step* function is generated for each node and where the program runs in an infinite loop that alternates reading inputs—calculating a step of the system and writing outputs. As it is adapted to both industrial certification and formal reasoning, this approach is followed by several implementations, such as SCADE Suite, Vélus, and other academic compilers. This is also the one taken here.

The architecture of the compiler is displayed in Figure 3. In the rest of the section, we describe the successive passes and

Figure 1. The Lustre code of the "counter" example and the corresponding machine code and C code.

```
node count () returns (out: bool)
var time: int;
let
  time = 0 -> if (pre time = 3)
        then 0 else pre time + 1;
  out = (time = 2);
tel
```

(a) Lustre code

```
node count () returns (out: bool)
var time, ptime: int; init, b: bool;
let
  init = true -> false;
  b = (ptime = 3);
  time = if init then 0 else
         if b then 0 else ptime + 1;
  out = (time = 2);
  ptime = pre time;
tel
```

(b) Normalized Lustre code

```
machine count {
  state ptime: int;
  instance: a: _arrow;

  step() returns (out: bool)
  var time: int; init, b: bool
  {
     b := state (ptime) = 3;
     init := a.step(true, false);
     if (init) {
       time := 0
     } else {
       if (b) {
          time := 0
       } else {
          time := state(ptime) + 1
       }
     }
     out := time = 2;
     state(ptime) := time;
  }
}
```

(c) Machine code

```
#define count_set_reset(self)\
  { self ->_reset = 1; }

void count_clear_reset(S *self) {
  if (self->_reset) {
     self->_reset = 0;
     _arrow_reset(self->a);
  }
}

void count_step(_Bool *out,
                S *self) {
  int time;
  _Bool init, b;
  count_clear_reset(self);
  b = self->ptime == 3;
  init = _arrow_step(self->a);
  if (init) {
     time = 0;
  } else {
     if (b) {
       time = 0;
     } else {
       time = self->ptime + 1;
     }
  }
  *out = time == 2;
  self->ptime = time;
}
```

(d) C code

Figure 2. Two representations of the execution of the example.

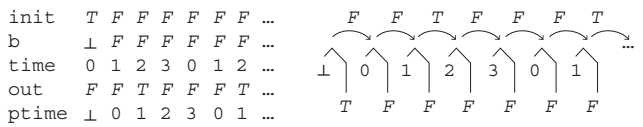


Figure 3. Architecture of the compiler.

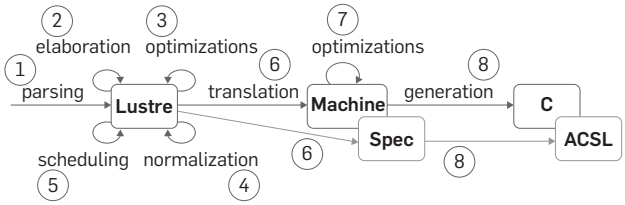


Figure 4. Normalized Lustre abstract syntax.

e	:=	**expression**
	c	constant
	x	variable
	$\diamond\ (\vec{e})$	operators
	e when $C(x)$	sampling
ck	:=	**clock**
	$\bullet$	base clock
	ck on $C(x)$	sub-clock
ce	:=	**control expression**
	e	expression
	if x then ce else ce	conditional
	merge $x(\overrightarrow{C \text{->} ce})$	merge
eq	:=	**equation**
	$x =_{ck} ce$	definition
	$x =_{ck}$ pre (e)	pre
	$\vec{x} =_{ck} f(\vec{e})$ [every x]	instantiation

present a formal definition of the involved languages. We skip the parsing, elaboration, and Lustre optimization steps; they are irrelevant to this work. We do not detail normalization and scheduling either, to simplify the presentation. We focus on steps 6, 7, and 8. In particular, the light grey boxes *Spec* and *ACSL* represent our main contribution. In addition to the regular generation of C code, we generate a specification encoding the semantics of the input Lustre nodes, attached to translated sequential code in the machine's intermediate language. This specification is then translated into ACSL and attached to the generated C code. This will be further developed in section 4.

3.2. Normalized lustre.

Normalization and scheduling are two source-to-source rewriting steps used to enable generation of imperative code. Normalization is used to identify and isolate state and stateful operations in dataflow nodes, by introducing auxiliary variables and equations to split complex expressions into simple sub-expressions. Scheduling is only a matter of re-ordering equations in preparation for the generation of sequential code. The ordering is based on a topological sort reflecting syntactic dependencies between variables.[4]

The abstract syntax of normalized Lustre is shown in Figure 4. In the remaining, we write $\vec{a}$ for the list $a_0 \cdots a_n$. The expression e when $C(x)$ is a *sampling* operation that describes the stream of e filtered at instants *when* the value of the variable x is equal to the enumerated type variant C. Such sampled substreams can be combined using the merge operator. These op-

erators highlight the notion of *clock*, that is, a boolean stream used to indicate when a computation is performed or not. The LustreC clock system follows the usual presentation from Colaço and Perez.[15] Succinctly, a clock is either the *base* clock (a stream that is always true) or a sub-clock (a sampled boolean stream). There are three forms of equations in normalized Lustre, each annotated with such a clock. Control and stateful operations appear at the top level, respectively through *definition* with a *control expression* and through *pre* and *node instantiation* (optionally with *modular reset* represented by the `every` keyword) equations. Modular reset[7] is a construct used to restart a node instance on some condition x.

Arrows get a special treatment. An expression e_1 -> e_2 is transformed into `if` *init* `then` *norm*(e_1) `else` *norm*(e_2), where *init* is defined by an additional equation *init* = `true->false`, that is, the stream that is always false but at the very first instant. In this equation, the arrow operation is considered as a node instantiation.

The normalized Lustre code of the counter example is presented on Figure 1b. Several local variables are introduced: `ptime` defines the previous value of `time, init` results from the normalization of the arrow operation, and `b` denotes the condition variable of the conditional.

3.3. Translation to machine code.

In the modular approach,[4] scheduled normalized Lustre code is translated into an intermediate imperative language with object-oriented features. Each Lustre node is translated into an object with an internal state and a method that executes one cycle of computation. The sequential statements of this *step* method are translated from the normalized and scheduled equations. The abstract syntax of the machine, our version of the language, is shown in Figure 5, and the translation function for expressions, control expressions, and equations is directly taken from Biernacki et al.[4]

Figure 1c presents the machine code translated from the example node. The variable `ptime`, defined by a `pre`, is transformed into a state variable (`state` keyword). The -> operation is transformed into a call to the *step* method of the corresponding sub-instance a (`instance` keyword; `_ arrow` is the name of the special machine that implements the behavior of the -> operation, considered as a special node instantiation). The step method is generated with the same signature as the node and comprises a sequence of statements directly translated from the Lustre equations.

Figure 5. Machine abstract syntax.

e	:=	**expression**
	c	constant
	x	variable
	`state` (x)	state variable
	$\diamond\ (\overrightarrow{e})$	operators
s	:=	**statement**
	$s; s$	sequence
	$x := e$	assignment
	`state` $(x) := e$	state assignment
	`if` (e) $\{s\}$ `else` $\{s\}$	conditionals
	`case` (e) $\{\overrightarrow{C:s}\}$	
	$x := i$.`step` $(\overrightarrow{e})$	step method call
	i.`reset()`	reset method call

3.4. Generation of C code.

The generation of C99-compliant C code is straightforward and follows once more the scheme described in Biernacki et al.[4] A structure is recursively generated for each machine, with fields for each state variable and each instance. The structure generated from the `count` example is shown below on the left, with the structure generated for the special machine `_arrow`.

```
struct _arrow { _Bool _first; };

typedef struct count_mem {
_Bool _reset ;
int ptime ;
struct _arrow_mem *a;
} S;
```

Generated fields for sub-instances are pointers to handle state update and separate compilation. A pointer to such a structure holding the state is passed to functions generated from machine methods.

We now explain the role of the field `_reset`. In Figure 1d, the `set _reset` macro is used to notify a sub-instance that it must be reset on the next cycle, by setting its `_reset` flag. The `clear _reset` function is called at the beginning of the step function: If the instance has to be reset (i.e., the `_reset` flag is true), then it actually reinitializes its arrow sub-instances and notifies its other node sub-instances for reset. Note that only one arrow sub-instance appears in this example.

The step method is transformed into a step function in a direct way. Outputs are passed by pointers to handle multiple outputs that are allowed in machine code. Each machine statement is transformed into a C statement. State variables and sub-instances are accessed through the `self` pointer to the state structure.

4. SEMANTICS AXIOMATIZATION

The original semantics for Lustre is the classic denotational dataflow semantics, where nodes are transformers of infinite streams as illustrated on the left side of Figure 2. Whereas on the right side, the state/transition operational semantics obtained by the compilation process described in section 3 feels very concrete. Unfortunately, axiomatizing stream transformers seems a rather difficult task, since every property must finally be expressed as mere C code assertions. Under the assumption it is possible, it is very likely that it would be inadequate or put too much stress on first-order back-end solvers used to discharge such assertions. Therefore, we choose to axiomatize instead a relational state/transition semantics, which lies in between. On the one hand, it is totally independent of the code optimizations described in section 5. On the other hand, it exposes a notion of state that is not part of the original semantics, yet state is simply made visible through normalization as explained in section 3.2, partially bridging the gap between our relational semantics and the dataflow one. We thus claim our semantics may perfectly serve as a reference semantics for Lustre. This kind of semantics also has the advantage of being easy to describe in a typed first-order logic with arithmetic[17] and is used internally by the Kind 2 Lustre model checker,[11] as well as by the Stc intermediate language of the Vélus compiler.[7]

Figure 6. Node semantics as a predicate.

$$
\begin{array}{l}
count\_tr\ (S, x, out, S') \triangleq \\
\quad \exists time, \\
\quad\quad S'(ptime) = time \\
\quad\quad \bigwedge\ out = (time = 2) \\
\quad\quad \bigwedge\ \exists init, b, \\
\quad\quad\quad\quad init \Rightarrow time = 0 \\
\quad\quad\quad\quad \bigwedge\ (\neg init \bigwedge\ b) \Rightarrow time = 0 \\
\quad\quad\quad\quad \bigwedge\ (\neg init \bigwedge\ \neg b) \Rightarrow \\
\quad\quad\quad\quad\quad time = S(ptime) + 1 \\
\quad\quad\quad\quad \bigwedge\ arrow\_tr\,(S[a], init, S'[a]) \\
\quad\quad\quad\quad \bigwedge\ b = (S(ptime) = 3)
\end{array}
$$

The semantics of a node can be represented as a relation that constrains input values, output values, a start state tree S, and an end state tree S'. The relation for the previous example is shown in Figure 6, where we write $S(ptime)$ for accessing the value of the state variable `ptime`, and $S[a]$ for accessing the sub-tree corresponding to the state of the arrow node instance. The scheduling of the variables—here $b \cdot init \cdot time \cdot out \cdot S'$—is used to build a more structured but equivalent predicate. The innermost bottom-up evaluation of the formula corresponds to the sequence of statements in the machine code. Quantifiers are introduced as soon as possible to tighten the scope of local variables. Our form (a) enables an incremental description of the transition relation, statement after statement, and therefore (b) allows verification tools to focus only on a local assertion context around each statement, as an efficient heuristic to discharge proof obligations entailed by the specification.

4.1. Formalization of flow equations semantics.

Each equation in the node is expressed as a constraint: definition and `pre` equations as equality constraints between variables (existentially quantified if they are local) where state variables are read in the start state S and written in the end state S', and node instantiations as corresponding transition relations constraining sub-states.

Figure 7 gives the formal state/transition semantics of normalized Lustre in first-order logic. The given definitions are parameterized by the states S and S' corresponding to the current node instance. The semantics functions resemble the translation functions from Lustre to machine code. A constant is evaluated to its value, a variable is mapped to either its symbol or to its access path in the start state S if it is a state variable, an operator application is recursively evaluated, and `when s` are again erased. Control-expression evaluation is parameterized by the variable being written and defined recursively: Conditional and merge expressions are turned into conjunctions of implications depending on the Boolean evaluation of the variable condition, with simple logical equations at their leaves (we write If a Then b Else c for $(a \Rightarrow b) \wedge (\neg a \Rightarrow c)$). The logical interpretation of an equation is wrapped by the $\mathcal{S}^{ck}$ function into a chain of implications that reflects the sub-clocking relations of its clock annotation ck. So, a definition equation is evaluated into an equation possibly nested in an implication; a pre-equation into an equation between the value of the state variable in the end state S' and the evaluation of its left-hand side; and a node instantiation into the evaluation of the corresponding transition relation

Figure 7. State/transition semantics of Lustre.

$$
\begin{array}{l}
[\![c]\!]_e = c \\
[\![x]\!]_e = c \begin{cases} S(x) & \text{if } x \text{ is def. by a } \texttt{pre}, \\ x & \text{otherwise.} \end{cases} \\
[\![\diamond(\vec{e})]\!]_e = \diamond(\overrightarrow{[\![e]\!]_e}) \\
[\![e \texttt{ when } C(x)\,(x)]\!]_e = [\![e]\!]^{y}_{ce} \\
[\![e]\!]^{y}_{ce} = (y = [\![e]\!]_e) \\
[\![\texttt{if } x \texttt{ then } ce_1 \texttt{ else } ce_2]\!]^{y}_{ce} = \text{If } x \text{ Then } [\![ce_1]\!]^{y}_{ce} \text{ Else } [\![ce_2]\!]^{y}_{ce} \\
[\![\texttt{merge } (\overrightarrow{C \texttt{ -> } ce})]\!]^{y}_{ce} = \bigwedge\ (x = C) \Rightarrow [\![ce]\!]^{y}_{ce} \\
[\![x =_{ck} ce]\!]_{eq} = \mathcal{S}^{ck}([\![ce]\!]^{x}_{ce}) \\
[\![x =_{ck} \texttt{pre } (e)]\!]_{eq} = \mathcal{S}^{ck}(S'(x) = [\![e]\!]_e) \\
[\![\vec{x} =_{ck} f(\vec{e})]\!]_{eq} = \mathcal{S}^{ck}(f\_tr(S[i], \overrightarrow{[\![e]\!]_e}, \vec{x}, S'[i])) \\
[\![\vec{x} =_{ck} f(\vec{e}) \texttt{ every } y]\!]_{eq} = \exists S_r, \\
\qquad \mathcal{S}^{ck}\begin{pmatrix} \text{If } y \text{ Then } f\_tr(S_r) \text{ Else } S_r = S[i] \\ \bigwedge f\_tr(S_r, \overrightarrow{[\![e]\!]_e}, \vec{x}, S'[i]) \end{pmatrix} \\
\qquad \mathcal{S}^{\bullet}(P) = P \\
\qquad \mathcal{S}^{ck \text{ on } C(x)}(P) = \mathcal{S}^{ck}((x = C \Rightarrow P)
\end{array}
$$

instantiated on the sub-states $S[i]$ and $S'[i]$. If there is a reset, the existential intermediate state S_r is reinitialized through $f\_rst(S_r)$; otherwise, it is equal to the start sub-state $S[i]$.

We define a relation f_tr_i for each equation eq_i, where n is the total number of equations in the node and $i \in [1, n]$, that builds the transition relation up to and including eq_i. This choice allows local reasoning relatively to each equation. We perform an analysis on the normalized and scheduled Lustre code that computes the set of *live variables* $\mathcal{L}_i$ for each equation eq_i. $\mathcal{L}_i$ is the set of assigned local or output variables so far, after the evaluation of eq_i, minus the set of local variables not occurring in the remaining equations $eq_{i+1}, \ldots, eq_n$. Last, we existentially quantify variables that were live before but not anymore after evaluation of eq_i.

A partial transition relation f_tr_i is associated to each equation, while the transition relation f_tr describes the whole node semantics.

$$
\begin{array}{rl}
f\_tr_i(S, \vec{I}, \vec{L_i}, \vec{O_i}, S') \triangleq & \exists \vec{V_i} f\_tr_{i-1}(S, \vec{I}, \overrightarrow{L_{i-1}}, \overrightarrow{O_{i-1}}, S') \\
& \wedge [\![eq_i]\!]_{eq} \\
f\_tr(S, \vec{I}, \vec{O}, S') \quad \triangleq & f\_tr_n(S_r, \vec{I}, \vec{O}, S')
\end{array}
$$

$\vec{I}$ are the input variables, $\vec{L_i}$ and $\vec{O_i}$ respectively are local and output variables that belong in $\mathcal{L}_i$. We define $\vec{V_i} = \overrightarrow{L_{i-1}} \backslash \vec{L_i}$, $f\_tr_0 = \top$, $\mathcal{L}_0 = \emptyset, \vec{L_n} = \emptyset$, and $\vec{O_n} = \vec{O}$.

4.2. Local annotations and function contracts.

Eventually the logical annotations attached to machine statements are translated into predicates, contracts, and as-

sertions in ACSL (ANSI/ISO C Specification Language). It supports primitives that cover the low-level aspects of C and that can be composed in a first-order logic. Through the Frama-C WP plug-in that implements a weakest precondition calculus, contracts and assertions can be checked by external SMT solvers, such as Alt-Ergo, CVC4 or Z3.

4.2.1. State representation.

To encode our transition relations, we first have to define a notion of *state*. Since ACSL supports C structures, we use a "flattened" version of the C structure that holds state as described in section 3.4. Sub-state is no longer referred by pointer, but directly included as a sub-structure. The structure is declared as *ghost*, meaning it can only be used in specification, not in the actual code. Below is the ghost structure generated for the `count` example.

```
/*@ ghost typedef struct count _ mem _ ghost {
            int ptime ;
            struct _ arrow _ mem _ ghost a;
          } gS;                          */
```

4.2.2. State correspondence.

We first assume that a standard initialization static analysis has been successfully performed on the Lustre input code, as it is common practice. It entails that every state variable m occurs in the right-hand side of an arrow instance ->, denoted by $Arrow(m)$, preventing that, at initial or reset time, its then unspecified value would be accessed.

To ensure that the ghost state stays in correspondence with the actual C state, we define a relation *f_pack* for each machine `f`, which in turn depends upon local versions f_pack_k, holding after each statement $s_k = \mathcal{T}_{\text{eq}}(eq_k)$. We denote by $Index(m)$ (resp. by $Index[i]$) the index k such that s_k assigns state variable m (resp. calls the step function of node instance i).

Let us suppose a machine `f` with n translated equations. We denote S_k the ghost state after equation eq_k. The C state is represented by the `self` pointer. In broad outline, *f_pack* recursively asserts that state variables values at the leaves of both ghost and actual trees are the same, provided protecting arrows are not in their initial state and *f* is not to be reset. Moreover, at locations after an arrow instance was called but before its state variables are updated, correspondence accounts for it by referring to this arrow at location 0, that is, prior to its call. This is the role of the r index computation in the following logical formulation, whose ACSL translation is not detailed. We write $\mathcal{I}_f$, resp. $\mathcal{S}_f$, for the sub-instances names, resp. state variables, of `f`.

$$f\_pack_k(S, \texttt{self}) \triangleq$$
$$\bigwedge_{i \in x_f} i\_pack(S[i], \texttt{self -> i})$$
$$\bigwedge_{m \in S_f} \neg arrow\_rst(S[Arrow(m)]_r) \Rightarrow S_k(m) = \texttt{self -> m}$$
$$\text{where } \begin{cases} r = 0 & \text{if } Index\,[Arrow\,(m)] \leq k < Index(m) \\ r = k & \text{otherwise} \end{cases}$$

$$f\_pack\,(S, \texttt{self}) \triangleq$$
$$\text{If } \texttt{self -> \_reset } \text{Then } f\_rst(S_n) \text{ Else } f\_pack_n\,(S, \texttt{self})$$

We also have to keep track of the C state assignments in our abstract state. To that purpose, we consider $\mathcal{G}(eq_i)$ statements as the ghost counterparts of $\mathcal{CC}(eq_i)$, the translation of the Lustre eq_i to C statements whenever they involve state variables. Otherwise, $\mathcal{G}(eq_i)$ is simply `skip`. We establish local simulation relations at each eq_i, used to compose a simulation at `step` function level. The relations constrain actual and ghost states of the C program. Figure 8 describes the corresponding simulation schemes. The scheme on the left represents a local simulation between the actual state in `self` and the partial ghost states S_i and S_{i+1}, after the execution of $\mathcal{CC}(eq_i)$ on one side and $\mathcal{G}(eq_i)$ on the other side: Memory correspondence is preserved, and the partial transition relation progresses one step further. The scheme on the right represents the combination of all such successive local simulations and is established at the step function level, between the actual state in `self` and the ghost start and end state S and S': Memory correspondence is preserved, and the transition relation is established.

4.2.3. Reset function contract.

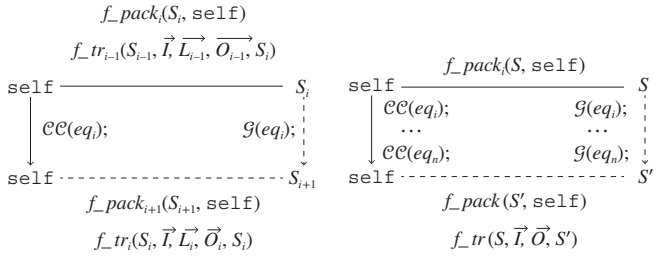
We add contract to the reset-related function described in section 3.4, as shown below for the example.

```
/*@ requires count _ pack (* mem , self );
    ensures count _ pack5 (* mem , self ); */
void count _ clear _ reset (S * self )
       /*@ ghost (gS \ghost *mem) */ {
  if (self -> _ reset ) {
   self -> _ reset = 0;
   _ arrow _ reset (self ->a);
  }
}
```

The contract for `count _ clear _ reset`, appearing as a special comment directly above function definition, states that memory correspondence is preserved, using `requires` and `ensures` keywords. While `self` is an actual parameter of the function, `mem` is declared as an additional ghost parameter.

Contrary to the compilation scheme we use for the reset, where actual recursive reinitialization is delayed until corresponding step calls on sub-states, we model abstract reinitialization in a direct "monolithic" way. To this end, we define a ghost function used to recursively reinitialize the ghost state in one take, displayed below here our example.

Figure 8. Fine- and coarse-grained simulation schemes.



```
/*@ ghost /@ ensures count _ reset (* mem ); @/
    void count _ reset _ ghost (gS \ghost *mem ) {
      _ arrow _ reset _ ghost (mem ->a);
      return ;
    }                                    */
```

The ghost function has a contract ensuring the state is indeed reinitialized, using an ACSL version of the *f_rst* predicate mentioned in section 4.1.

4.2.4. Step function contract.
Partial transition-relations definitions are readily translated into ACSL predicates as relations between two ghost states corresponding respectively to S and S'. We then generate a contract for the `step` function, and each annotation is translated into an ACSL assertion. Stateful operations are reflected on the ghost state using ghost statements. The instrumented code of the generated `step` function for the example is displayed below. We omit the definition of the generated ACSL predicates for each *count_tr*$_i$.

```
/*@ requires count _ pack (* mem , self );
    ensures count _ pack (* mem , self );
    ensures count _ tr
        ( \old (* mem), x, *out , *mem ); */
void count _ step ( _ Bool *out , S * self )
                /*@ ghost (gS \ghost *mem) */ {
  int time ;
  _ Bool init , b;
  count _ clear _ reset ( self )/*@ ghost ( mem)*/;
  //@ assert count _ tr0
        ( \at (* mem , Pre), x, * mem );
  b = (self -> ptime == 3);
  //@ assert count _ tr1
        ( \at (* mem , Pre), x, b, * mem );
  init = _ arrow _ step
        (self ->a)/*@ ghost (& mem ->a) */;
  //@ assert count _ tr2
        ( \at (* mem , Pre), x, b, init ,
  //@                  *mem );
  if ( init ) { time = 0; } else {
    if (b) { time = 0; } else {
      time = self -> ptime + 1;
    }
  }
  //@ assert count _ tr3
        (\at (* mem , Pre), x, time ,
  //@                  *mem );
  *out = ( time == 2);
  //@ assert count _ tr4
        (\at (* mem , Pre), x, time , *out ,
  //@                  *mem );
  self -> ptime = time ;
  //@ ghost mem -> ptime = time ;
  //@ assert count _ tr5
        (\at (* mem , Pre), x, *out ,
  //@                  *mem );
}
```

The contract requires that the state correspondence holds before the call, and ensures it is preserved after. Moreover, it states that the transition relation holds between the ghost state before the call and the ghost state after, ensuring the correctness result: the C code respects the semantics of the node. The terms `\old(*mem)` in the contract and `\at(*mem, Pre)` in the assertions both refer to the value of `*mem` before the call of the function. In practice, we also generate assertions enabling the establishment of the memory correspondence at each intermediate program point.

5. OPTIMIZATIONS AND PROOFS

First, simply notifying reset at C code level instead of actually performing it is already a supported optimization that does not go unnoticed when running Lustre state machines. This is also the way the SCADE suite handles node resetting.

We detail this in the following several other optimizations that LustreC supports. Since these optimizations may replace or erase variables, and even modify the machine statements themselves, we must take care of the partial transition relations that annotate them. Whereas *f_tr*$_i$ and *f_tr* keep the same definitions, the actual parameters $\vec{L_i}$ of *f_tr*$_i$ $\left(S,\vec{I},\vec{L_i},\vec{O_i},S'\right)$ may change according to the optimization level. Also, moving annotations around may yield capture problems. There are several ways of handling those issues, for example, involving existential quantification, but we choose to rely instead on so-called *ghost variables*. Ghost variables are simply variables that can only be used in the specification but not in the actual executable code. Hence, it means the semantics encoding generated when producing unoptimized machine code is unchanged by further optimizations. We describe the effects of the different optimizations applied to the source Lustre toy example presented in Figure 9a, which underlines the use of user-defined enumerated types as clocks. Figure 9b is the generated machine code without any optimization. We represent annotations as special `--@ f _ tr _ i(...)` comments, where the partial transition relations *f_tr*$_i$ are defined as described in the previous section (without the S and S' parameters since they are irrelevant to optimizations), and introduced assignments to ghost variables are written `--@ x := e`. Figure 9c presents the fully optimized machine code, and the four main optimizations are presented in the following.

Conditionals fusion. Without further transformations, two adjacent equations with the same sub-clock are transformed into two adjacent conditional statements guarded on the same condition. A typical optimization that Lustre compilers following the modular approach implement is a rewriting pass that fuses such groups of conditionals. Extending readily,[4] implementation consists in merging adjacent conditional branches and regrouping their annotations. We can see in Figure 9c the high number of generated conditionals fused to produce better code, particularly on the conditionals concerning the `d` variable.

Variable inlining. Variable inlining occurs only when its defining expression is atomic. Thus, substituting this expression for the variable does not duplicate complex expression evaluation. Such substitutions are performed in code only. The annotations are untouched, since the defining statement is turned into a ghost one so that the inlined variable is

Figure 9. Lustre example with non-optimized and optimized translated machine code.

```
type en1 = enum { On, Off };
type en2 = enum { Up, Down };

node clocks (x: int) returns (y: int)
var c: en1 clock; d: en2 clock;
        b1,b2,b3,z: int; c1,c2: bool

let
   c1 = (x >= 0);
   d =   if c1 then Up else Down;
   c2 =  (x = 0) when Up(d)
   c = if c2 then Off else On;
   b2 = 2 when Off(c);
   b1 =  1 when On(c);
   z = merge c (On -> b1) (Off -> b2);
   b3 = 3 when Down(d);
   y = merge d (Up -> z) (Down -> b3);
tel
```

(a) Lustre code

```
step(x: int) returns (y: int)
var c: en1; d: en2; b1,b2,b3,z: int; c1,c2: bool
{
   c1 := x >= 0;
   --@ clocks_tr1(x, c1)
   if (c1) { d := Up } else { d := Down }
   --@ clocks_tr2(x, d)
   case (d) { Up: c2 := x = 0 }
   --@ clock_tr3(x, d, c2)
   case (d) {
     Up: if (c2) { c := Off } else { c := On } }
   --@ clocks_tr4(x, d, c)
   case (d) { Up: case (c) { Off: b2 := 2 } }
   --@ clock_tr5(x, d, c, b2)
   case (d) { Up: case (c) { On: b1 := 1 } }
   --@ clocks_tr6(x, d, c, b1, b2)
   case (d) { Up: cas (c) { On: z := b1
                            Off: z := b2 } }

   --@ clocks_tr7(x, d, z)
   case (d) { Down: b3 := 3 }
   --@ clocks_tr8(x, d, b3, z)
   case (d) { Up: y := z
              Down: y := b3 }
   --@ clocks_tr9(x, y)
}
```

(b) Machine code

```
step(x: int) returns (y: int)
{
   --@ c1 := x >= 0
   --@ clocks_tr1(x, c1)
   if (x >= 0) {
     --@ d := Up
     --@ c2 := x = 0
     if (x = 0) {
        --@ c := Off
        --@ b2 := 2
        y := 2
        --@ z := y
     } else {
        --@ c := On
        --@ b1 := 1
        y := 1
        --@ z := y
     }
   } else {
     --@ d := Down
     --@ b3 := 3
     y := 3
   }
   --@ clocks_tr2(x, d)
   --@ clocks_tr3(x, d, c2)
   --@ clocks_tr4(x, d, c)
   --@ clocks_tr5(x, d, c, b2)
   --@ clocks_tr6(x, d, c, b1, b2)
   --@ clocks_tr7(x, d, z)
   --@ clocks_tr8(x, d, b3, z)
   --@ clocks_tr9(x, y)
}
```

(c) Full optimization

kept alive in the specification. In Figure 9c, variables `b1`, `b2`, `b3`, `c1`, and `c2` are inlined in the statements but turned into ghost variables in the specification.

Variable recycling. We exploit variable reuse, applied only between variables of the same type, for the sake of safety and traceability. We leverage the results of liveness analysis and clock calculus to reuse dead variables or clock-disjoint ones, that is, variables that cannot simultaneously bear meaningful values in the same time frame. As for variable inlining, the variable replaced by a reused one is turned into a ghost variable to keep its original definition in the specification. However, because code after this optimization is no longer in static single-assignment (SSA) form, capture problems may arise when annotations refer to a variable that has been reused. To deal with such issues, we introduce for each variable y, which will later be reused a ghost alias y' assigned only once with the original defining value of y. In subsequent annotations, y' is substituted for y. On the example in Figure 9c, only the variable `z` is replaced by `y`. The aforementioned capture problem does not arise here and there is no need to introduce a ghost alias y'.

Enumerated type elimination. For a variable x belonging in an enumerated type (for example, a clock), the compiler merges conditional assignment of x to enumeration constants with a conditional statement depending upon x. This proves useful for clock-heavy programs obtained from Lustre state machines. We again address potential capture problems by turning variable x into a ghost variable. We can see in the code in Figure 9c that variables `c` and `d` have been eliminated. As a result, switch cases are merged accordingly and each variable is kept as a ghost in the specification.

6. EXPERIMENTAL RESULTS

To evaluate our compiler extension, we ran it against a set of Lustre programs taken from the benchmarking suite of the Kind tool,[17] and from the test suite of the CoCoSim tool.[6] We used the default level of optimization of the compiler (O2), that is conditionals fusion and variable inlining. The tests were run on a machine equipped with two Intel® Xeon® processors E5-2670 v3 @ 2.30 GHz with 12 cores (24 threads) each and 64 GB RAM. Frama-C / WP 26.1 is run with a global timeout of 15360 s, using the Alt-Ergo 2.4.2, CVC4 1.8 and Z3 4.11.2 solvers in parallel, with a timeout per individual proof obligation (PO) of 60 s.

Figure 10. Experiments report with O2 optimization level.

	Generated	Verified	Verified (%)
Files	399	370	92.73
POs	231,471	231,210	99.89

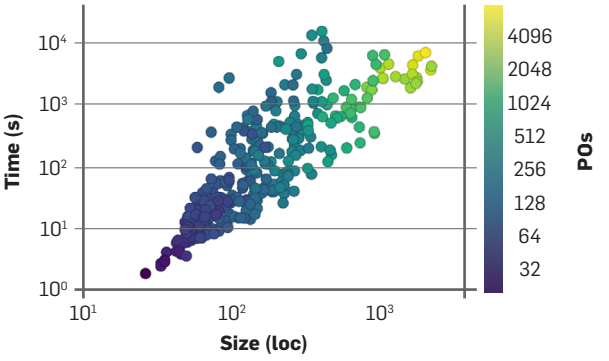


Figure 10 presents a summary table and a scattered log-log plot displaying the distribution of the verification time of these test files against the size of the generated C code (ignoring ACSL specification): It roughly outlines a linear distribution. The number of generated POs per file, displayed following the color scale, is also linear with regard to code size. The conference version of this paper presents more detailed results for the interested reader.[8]

7. DISCUSSION

We succeeded in automatically providing to each Lustre source code an abstract operational semantics whose preservation can be proved with high success rate at the C target level of a Lustre compiler. We achieved our goal with a translation validation technique, on a non trivial subset of the Lustre language while enabling code optimizations. To the best of our knowledge, the most aggressive of our optimizations, such as clock disjoint time-frame variable recycling, are not supported by the state-of-the-art SCADE Suite compiler. The automated support for such strong specification of C code also allowed us to unveil a bug in the original LustreC compiler optimization strategies.

Building on this promising first proposal, our work can be extended in several directions. First, we need to investigate how to increase efficiency and robustness of the solvers, by providing aggressive context pruning techniques and guidance to these tools. We may for instance reconsider our position detailed in section 4.2 about state correspondence once the Frama-C tool supports local reasoning again. Also, even though using ghost variables instead of existential quantification as explained in section 5 probably helps solvers by keeping the exact same code and annotations structure whatever the optimization level, we may try a different balance between these two approaches. We also want to find a more suitable metric than program size to sort out the several ways of improving our verification approach, such as depth or size of the state tree.

Second, we could provide support for a more expressive input language, including for instance structured datatypes such as records and arrays. Until now, we also assume that Lustre programs are well-formed, i.e. free of run-time errors and uninitialized variables, otherwise such programs simply cannot be proved to follow their specification. We may investigate what remains of their specification when well-formedness does not hold.

Finally, with regard to our relational semantics, we plan to address its relationship with the canonical dataflow one and envision initiating another approach based upon a formalization in a proof assistant such as Coq,complemented with automated proof strategies, instead of putting heavy stress on first-order solvers. We also plan to use it to prove high-level functional contracts of Lustre programs.

8. ACKNOWLEDGMENTS

This work is supported by the Defense Innovation Agency (AID) of the French Ministry of Defense (research project CLEDESCHAMPS N 2021 65 0070) and by JST CREST Grant Number JPMJCR21M3.

References

1. Amjad, H.M. et al. Translation validation of code generation from the SIGNAL Data-Flow Language to Verilog. In *Proc. 2019 15th Intern. Conf. on Semantics, Knowledge and Grids*, 153–160.
2. Baudin, P. et al. The dogged pursuit of bug-free C programs: The Frama-C software analysis platform. *Commun. ACM 64*, 8 (July 2021), 56–68.
3. Benveniste, A. and Le Guernic, P. Hybrid dynamical systems theory and the Signal language. *IEEE Trans. on Automatic Control 35*, 5 (May 1990), 535–546.
4. Biernacki, D., Colaço, J.-L., Hamon, G., and Pouzet, M. Clock-directed modular code generation for synchronous data-flow languages. In *Proc. of the 2008 ACM SIGPLAN-SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems*, ACM, 121–130.
5. Bohrer, B. et al. VeriPhy: Verified controller executables from verified cyber-physical system models. In *Proc. of the 39th ACM SIGPLAN Conf. on Programming Language Design and Implementation*, ACM (2018), 617–630.
6. Bourbouh, H. et al. CoCoSim, a code generation framework for control/command applications: An overview of CoCoSim for multi-periodic discrete Simulink models. In *Embedded Real Time Systems 2020*.
7. Bourke, T., Brun, L., and Pouzet, M. Mechanized semantics and verified compilation for a dataflow synchronous language with reset. In *Proc. of the 47th ACM SIGPLAN Symp. on Principles of Programming Languages 4*, ACM (2020), 29.
8. Brun, L., Garion, C., Garoche, P.-L., and Thirioux, X. Equation-directed axiomatization of Lustre semantics to enable optimized code validation. *ACM Trans. on Embedded Computing Systems 22*, 5s (Sept. 2023), 151:1–151:24.
9. Caspi, P., Pilaud, D., Halbwachs, N., and Plaice, J. A. LUSTRE: A declarative language for programming synchronous systems. In Proc. *14th Symp. on Principles of Programming Languages. ACM*, POPL'87, 1987.
10. Cavalcanti, A., Clayton, P., and O'Halloran, C. From control law diagrams to Ada via Circus. *Formal Aspects of Computing 23*, 4 2011, 465–512.
11. Champion, A., Mebsout, A., Sticksel, C., and Tinelli, C. The Kind 2 model checker. In *Proc. of the 28th Intern. Conf. on Computer Aided Verification 9780*, Chaudhuri, S. and Farzan, A. eds, Springer (2016), 510–517.
12. Chan Ngo, V., Talpin, J.-P., and Gautier, T. Translation validation for synchronous data-flow specification in the SIGNAL compiler. *Formal Techniques for Distributed Objects, Components, and Systems*, Lecture Notes in Computer Science, S. Graf and M. Viswanathan, eds., Springer, Cham, (2015), 66–80.
13. Chen, M. et al. MARS: A toolchain for modelling, analysis and verification of hybrid systems. *Provably Correct Systems*, NASA Monographs in Systems and Software Engineering, M. Hinchey, J.P. Bowen, and E.-R. Olderog, eds. Springer, Cham, (2017), 39–58.
14. Colaço, J.-L., Pagano, B., and Pouzet, M. SCADE 6: A formal language for embedded critical software development. In Proc. of the *2017 Intern. Symp. on Theoretical Aspects of Software Engineering*, 2017, 1–11.
15. Colaço, J.-L. and Pouzet, M. Clocks as first class abstract types. In *Embedded Software*, Lecture Notes in Computer Science, Springer (2003), 134–155.
16. Finkbeiner, B., Oswald, S., Passing, N., and Schwenger, M. Verified Rust monitors for Lola specifications. In *Proc. of the Runtime Verification: 20th Intern. Conf.*, Springer-Verlag, (October 2020), 431–450.
17. Hagen, G. and Tinelli, C. Scaling up the formal verification of Lustre programs with SMT-based techniques. In *Proceedings of the 2008 Intern. Conf. on Formal Methods in Computer-Aided Design*, IEEE Press, 1–9.
18. Leroy, X. Formal verification of a realistic compiler. *Commun. ACM 52*, 7 2009, 107–115.
19. Pnueli, A., Siegel, M., and Singerman, E. Translation validation. *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, B. Steffen, ed. Springer (1998), 151–166.
20. Pnueli, A., Strichman, O., and Siegel, M. Translation validation for synchronous languages. In *Proc. of the 25th Intern. Colloquium on Automata, Languages and Programming*, Springer-Verlag (1998), 235–246.
21. Pnueli, A., Strichman, O., and Siegel, M. Translation validation: From SIGNAL to C. *Correct System Design: Recent Insights and Advances*, Lecture Notes in Computer Science, E.-R. Olderog, E.-R. and Steffen, B., eds., Springer, (1999) 231–255.
22. Ryabtsev, M. and Strichman, O. Translation validation: From Simulink to C. *Computer-Aided Verification, Lecture Notes in Computer Science*, A. Bouajjani and O. Maler, eds., Springer, Berlin, Heidelberg, (2009), 696–701.
23. Shi, G. et al. A formally verified transformation to unify multiple nested clocks for a Lustre-like language. *Science China Information Sciences 62*, 1 (2019), 12801.
24. Toom, A. et al. Towards reliable code generation with an open tool: Evolutions of the Gene-Auto toolset. In *ERTS2 2010, Embedded Real Time Software & Systems* Toulouse, France.
25. Yang, Z. et al. Towards a verified compiler prototype for the synchronous language SIGNAL. *Frontiers of Computer Science 10*, 1 (2016), 37–53.

Steven Christophe Garion, ISAE-SUPAERO, University of Toulouse, Toulouse, France.

Lélio Brun, National Institute of Informatics, Tokyo, Japan.

Pierre-Loïc Garoche, ENAC LII, Toulouse, France.

Xavier Thirioux, ISAE-SUPAERO, University of Toulouse, Toulouse, France.