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

LÉLIO BRUN, National Institute of Informatics, Japan
CHRISTOPHE GARION, ISAE-SUPAERO, University of Toulouse, France
PIERRE-LOÏC GAROCHE, ENAC, University of Toulouse, France
XAVIER THIRIOUX, ISAE-SUPAERO, University of Toulouse, France

Model-based design tools like SCADE Suite and Simulink 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.

CCS Concepts: • **Computer systems organization → Real-time languages**; *Embedded systems*; • **Software and its engineering → Compilers**; **Source code generation**; **Formal software verification**; **Model-driven software engineering**; **Data flow languages**.

Additional Key Words and Phrases: Lustre, Frama-C, ACSL

ACM Reference Format:
Lélio Brun, Christophe Garion, Pierre-Loïc Garoche, and Xavier Thirioux. 2023. Equation-Directed Axiomatization of Lustre Semantics to Enable Optimized Code Validation. *ACM Trans. Embedd. Comput. Syst.* 1, 1 (September 2023), 24 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Model-based design tools like SCADE Suite [2] or Simulink [33] 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 [6] such as Lustre [17]. Lustre 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 [22] and can encode a subset of the discrete part of Simulink [38, 16, 45, 10].

Languages of the dataflow synchronous family usually share well studied formal semantics and compilation techniques, allowing traceability, industrial certification and verification. In the

Authors' addresses: Lélio Brun, National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, 101-8430, Japan, lelio_brun@nii.ac.jp; Christophe Garion, ISAE-SUPAERO, University of Toulouse, 10, avenue Édouard-Belin, Toulouse, 31055, France, garion@isae-supaero.fr; Pierre-Loïc Garoche, ENAC, University of Toulouse, 7, avenue Édouard-Belin, Toulouse, 31055, France, garoche@enac.fr; Xavier Thirioux, ISAE-SUPAERO, University of Toulouse, 10, avenue Édouard-Belin, Toulouse, 31055, France, thirioux@isae-supaero.fr.

1539-9087/2023/9-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

domain of safety-critical embedded software design, these features are paramount to ensure 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 [12, 11] whose correctness is verified once and for all in Coq [41], we choose another approach to verified compilation: translation validation [34]. 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 an existing Lustre to C compiler, LustreC [42], 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 [5] 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 [8] which prevails in modern Lustre code generators such as SCADE Suite. While some Lustre model-checking tools [28, 19] 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 rather 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,
- associated to each generated instruction, enabling automatic validation,
- and that is compatible with several optimizations at code level.

The paper is organized as follows. Section 2 presents an overview of related works. Section 3 gives a description of 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 for synchronous languages. The goal of the GeneAuto project [43, 44] was to develop a qualified code generator for a subset of Simulink, with parts proved in Coq. Some preliminary work [46] showed semantics preservation results for some passes of a compiler for the Signal language [7]. To our knowledge the more advanced solutions focus on Lustre: [40, 39] give an end-to-end correctness proof from an imperatively defined dataflow semantics to the semantics of C, while the Vélus compiler [12, 11] uses a stream-based dataflow semantics and is built on the verified C compiler CompCert [32]. These solutions are proof-of-concepts 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 re-prove systematically the compiler when a variation is made in the compilation process. Indeed, LustreC is a rather large software with about 40000 lines of OCaml code, as it is designed as an experimental playground for Lustre compilation,

with several additional features. On the other hand, Vélus is equally large with about 40000 lines of Coq code, but the extracted code used to build the actual compiler only amounts to about 1500 lines of OCaml code [14]. This comparison highlights the fact that the two approaches actually aim for different goals. Vélus is an experimental proof that 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 the verification of 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, i.e. 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 is an approach that was early applied to synchronous languages [35]. 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, 20, 36] and on Simulink [18, 37]. In particular, [18] proposes a framework to show refinement relations between Simulink discrete-time block diagrams and SPARK / Ada implementations. These works and our solution, that specifically targets compilation from Lustre to C, are in the same vein. The authors of [27] follow essentially the same approach than 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 miminizing the memory footprint of generated monitors. Both [18] and [27] handle a rather simple input language, lacking advanced control structures such as clock sampling, resetting and state machines. Furthermore, it seems that 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 put emphasis on 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 tackle specifically design and verification of hybrid systems. The MARS [21] framework extends [47, 48] to provide an integrated solution to design and verify hybrid Simulink models. Several rewriting steps are used and verification is performed by simulation. VeriPhy [9] 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 AND ITS COMPILATION

We present the Lustre language on the simple example below.

```
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
```

init	T	F	F	F	F	F	F	$\cdots$
b	$\bot$	F	F	F	T	F	F	$\cdots$
time	0	1	2	3	0	1	2	$\cdots$
out	F	F	T	F	F	F	T	$\cdots$
ptime	$\bot$	0	1	2	3	0	1	$\cdots$

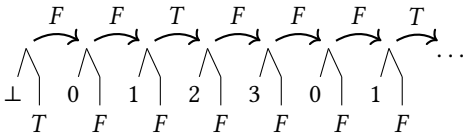
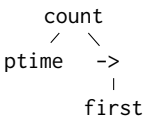


Fig. 1. Two representations of the execution of the example

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 streams `b`, `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 `init` is defined with the `->` operator: it has the value T (`true`) at the initial instant and the value F (`false`) otherwise. Finally, the stream `ptime` is defined with the **`pre`** operator that represents an uninitialized delay.

Two ways of representing the execution are shown in fig. 1. On the dataflow representation on the left, each variable is associated to its corresponding stream. The columns give the values of the streams indexed at a each successive instant. We can clearly describe the behavior of the **`pre`** operator: the stream associated with `ptime` is the stream associated with `time` delayed by one instant, where $\bot$ 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. The successive transitions, labeled with the indexed output values, encode the node equations.



The state of the `count` node is represented as the tree on the left and is comprised of the *state variable* `ptime`, the only variable defined by a unit delay (**`pre`** operator), and by a sub-tree corresponding to the `->` operator. Indeed, we consider this operator as a special node with its own state variable `first`.

3.1 Compiler architecture

The standard Lustre compilation approach, described in [8], consists in 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 like SCADE Suite, Vélus, and other academic compilers [31, 26]. This is also the one that is taken here.

The architecture of the compiler is displayed in fig. 2. In the remaining of the section, we describe the successive passes and present a formal definition of the involved languages. We skip the parsing, elaboration and Lustre optimization steps since they are irrelevant to this work. We also do not

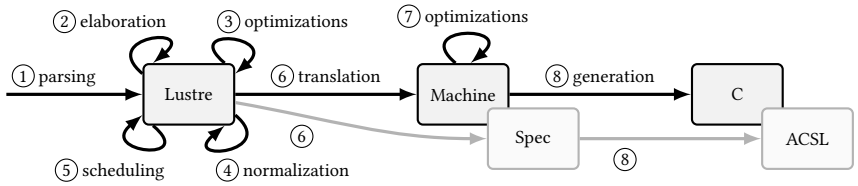


Fig. 2. Architecture of the compiler

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

Fig. 3. Normalized Lustre abstract syntax

detail normalization and scheduling, to simplify the presentation. Hence, we will focus in the following on the 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 intermediate language. This specification is then translated into ACSL, the specification language of the Frama-C platform, 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 as described, e.g. in [8].

The abstract syntax of normalized Lustre is shown in fig. 3. In the remaining, we write $\overrightarrow{a}$ 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 sub-streams can be combined using the **merge** operator. These operators highlight the notion of *clock*, i.e. a boolean stream used to indicate when a computation is performed or not. The LustreC clock system follows the usual presentation from [23]: 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 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 [30, 12] is a construct used to restart a node instance on some condition x.

3.3 Translation to Machine code

In the modular approach [8], 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 Machine, our version of the language, is shown in fig. 4, and the translation function for expressions, control expressions and equations is summarized in fig. 5. Expression translation $\mathcal{T}_e()$ is straightforward: a constant becomes a constant; a variable is turned into either a

$$
\begin{array}{lr}
e := & \textbf{expression} \\
\quad \mid\ c & \text{constant} \\
\quad \mid\ x & \text{variable} \\
\quad \mid\ \texttt{state}(x) & \text{state variable} \\
\quad \mid\ \diamond(\overrightarrow{e}) & \text{operators}
\end{array}
\qquad
\begin{array}{lr}
s := & \textbf{statement} \\
\quad \mid\ s; s & \text{sequence} \\
\quad \mid\ x := e & \text{assignment} \\
\quad \mid\ \texttt{state}(x) := e & \text{state assignment} \\
\quad \mid\ \texttt{if}\,(e)\,\{\,s\,\}\,\texttt{else}\,\{\,s\,\} & \text{conditionals} \\
\quad \mid\ \texttt{case}\,(e)\,\{\ \overrightarrow{C\colon s}\,\} & \\
\quad \mid\ \overrightarrow{x} := i.\texttt{step}(\overrightarrow{e}) & \text{step method call} \\
\quad \mid\ i.\texttt{reset}() & \text{reset method call}
\end{array}
$$

Fig. 4. Machine abstract syntax

$$
\begin{aligned}
\mathcal{T}_e(c) &= c \\
\mathcal{T}_e(x) &= \begin{cases} \texttt{state}(x) & \text{if } x \text{ is defined by a } \textbf{pre}, \\ x & \text{otherwise.} \end{cases} \\
\mathcal{T}_e\left(\diamond(\overrightarrow{e})\right) &= \diamond(\overrightarrow{\mathcal{T}_e(e)}) \\
\mathcal{T}_e\left(e\ \textbf{when}\ C(x)\right) &= \mathcal{T}_e(e) \\
\\
\mathcal{T}^y_{ce}(e) &= y := \mathcal{T}_e(e) \\
\mathcal{T}^y_{ce}\left(\textbf{if}\ x\ \textbf{then}\ ce_1\ \textbf{else}\ ce_2\right) &= \texttt{if}\,(x)\,\{\,\mathcal{T}^y_{ce}(ce_1)\,\}\,\texttt{else}\,\{\,\mathcal{T}^y_{ce}(ce_2)\,\} \\
\mathcal{T}^y_{ce}\left(\textbf{merge}\ x\ \overrightarrow{(C\ \texttt{->}\ ce)}\right) &= \texttt{case}\,(x)\,\{\ \overrightarrow{C\colon \mathcal{T}^y_{ce}(ce)}\,\} \\
\\
\mathcal{T}_{eq}(x =_{ck} ce) &= C^{ck}\left(\mathcal{T}^x_{ce}(ce)\right) \\
\mathcal{T}_{eq}(x =_{ck} \textbf{pre}(e)) &= C^{ck}\left(\texttt{state}(x) := \mathcal{T}_e(e)\right) \\
\mathcal{T}_{eq}\left(\overrightarrow{x} =_{ck} f(\overrightarrow{e})\right) &= C^{ck}\left(\overrightarrow{x} := i.\texttt{step}(\overrightarrow{\mathcal{T}_e(e)})\right) \quad \text{where } i \text{ is fresh} \\
\mathcal{T}_{eq}\left(\overrightarrow{x} =_{ck} f(\overrightarrow{e})\ \textbf{every}\ y\right) &= C^{ck}\left(\begin{array}{l}\texttt{if}\,(y)\,\{\,i.\texttt{reset}()\,\};\\ \overrightarrow{x} := i.\texttt{step}(\overrightarrow{\mathcal{T}_e(e)})\end{array}\right) \quad \text{where } i \text{ is fresh} \\
\\
C^{\bullet}(s) &= s \\
C^{ck\ \textbf{on}\ C(x)}(s) &= C^{ck}\left(\texttt{case}\,(x)\,\{\ C\colon s\,\}\right)
\end{aligned}
$$

Fig. 5. Translation function from Lustre to Machine

simple variable or a state variable if it is defined using a **pre**; an operator application is recursively translated; and **when** s are simply erased because the clock behavior is handled at the level of equations, as explained in the following. Control expression translation $\mathcal{T}^y_{ce}()$, parameterized by the variable y being written, is defined recursively: conditional and merge expressions are turned into conditional statements, with assignments at their leaves. The statement resulting from the translation of an equation annotated with clock ck is wrapped by the $C^{ck}()$ function in possibly nested conditionals, in order to implement the control structure that the clock calculus describes. Hence, a definition equation is turned into an assignment; a pre equation into a state assignment; and a node instantiation into a step method call, optionally preceded with a reset method call in case of reset. The instance name i is uniquely generated and designates the object associated with this particular instantiation of the node f.

Figure 6 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 is comprised of a sequence of statements directly translated from the Lustre equations.

3.4 Generation of C code

The generation of C99 compliant C code is rather straightforward and follows again the scheme described in [8]. 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_mem { _Bool _first; };

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

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 fig. 7 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 fig. 1. 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 backend 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 has also the advantage of being easy to describe in a typed first-order logic with arithmetic [29] and is used internally by the Kind 2 Lustre model checker [19], and also by the Stc intermediate language of the Vélus compiler [12].

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 fig. 8, 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 fig. 8a corresponds

```
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;
  }
}
```

Fig. 6. 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;
}
```

Fig. 7. C code

$$
\begin{aligned}
&count\_tr(S, x, out, S') \triangleq \\
&\quad \exists time,\ init,\ b, \\
&\qquad S'(ptime) = time \\
&\qquad \wedge\ b = (S(ptime) = 3) \\
&\qquad \wedge\ arrow\_tr(S[a], init, S'[a]) \\
&\qquad \wedge\ init \implies time = 0 \\
&\qquad \wedge\ (\neg init \wedge b) \implies time = 0 \\
&\qquad \wedge\ (\neg init \wedge \neg b) \implies \\
&\qquad\quad time = S(ptime) + 1 \\
&\qquad \wedge\ out = (time = 2)
\end{aligned}
$$

(a) As a conjunction.

$$
\begin{aligned}
&count\_tr(S, x, out, S') \triangleq \\
&\quad \exists time, \\
&\qquad S'(ptime) = time \\
&\qquad \wedge\ out = (time = 2) \\
&\qquad \wedge\ \exists init,\ b, \\
&\qquad\quad init \implies time = 0 \\
&\qquad\quad \wedge\ (\neg init \wedge b) \implies time = 0 \\
&\qquad\quad \wedge\ (\neg init \wedge \neg b) \implies \\
&\qquad\qquad time = S(ptime) + 1 \\
&\qquad\quad \wedge\ arrow\_tr(S[a], init, S'[a]) \\
&\qquad\quad \wedge\ b = (S(ptime) = 3)
\end{aligned}
$$

(b) As a composition.

Fig. 8. Node semantics as a predicate.

to a typical predicate encoding the node semantics, as produced by Lustre model-checking tools [28, 19]. This prenex normal form predicate describes the logical relationship between the state before and after the transition, and between the input and the output variables. All local variables are existentially quantified. In our proposal, displayed in fig. 8b, 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,

$$
\begin{aligned}
\llbracket c \rrbracket_e &= c \\
\llbracket x \rrbracket_e &= \begin{cases} S(x) & \text{if } x \text{ is defined by a } \textbf{pre}, \\ x & \text{otherwise.} \end{cases} \\
\llbracket \diamond(\overrightarrow{e}) \rrbracket_e &= \diamond(\overrightarrow{\llbracket e \rrbracket_e}) \\
\llbracket e \textbf{ when } C(x) \rrbracket_e &= \llbracket e \rrbracket_e \\
\\
\llbracket e \rrbracket^y_{ce} &= (y = \llbracket e \rrbracket_e) \\
\llbracket \textbf{if } x \textbf{ then } ce_1 \textbf{ else } ce_2 \rrbracket^y_{ce} &= \text{If } x \text{ Then } \llbracket ce_1 \rrbracket^y_{ce} \text{ Else } \llbracket ce_2 \rrbracket^y_{ce} \\
\left\llbracket \textbf{merge } x \overrightarrow{(C \text{ -> } ce)} \right\rrbracket^y_{ce} &= \bigwedge \overrightarrow{(x = C) \implies \llbracket ce \rrbracket^y_{ce}} \\
\\
\llbracket x =_{ck} ce \rrbracket_{eq} &= \mathcal{S}^{ck}\left(\llbracket ce \rrbracket^x_{ce}\right) \\
\llbracket x =_{ck} \textbf{pre}(e) \rrbracket_{eq} &= \mathcal{S}^{ck}\left(S'(x) = \llbracket e \rrbracket_e\right) \\
\llbracket \overrightarrow{x} =_{ck} f(\overrightarrow{e}) \rrbracket_{eq} &= \mathcal{S}^{ck}\left(f\_tr\left(S[i], \overrightarrow{\llbracket e \rrbracket_e}, \overrightarrow{x}, S'[i]\right)\right) \\
\llbracket \overrightarrow{x} =_{ck} f(\overrightarrow{e}) \textbf{ every } y \rrbracket_{eq} &= \exists S_r, \mathcal{S}^{ck}\begin{pmatrix} \text{If } y \text{ Then } f\_rst(S_r) \text{ Else } S_r = S[i] \\ \wedge\, f\_tr\left(S_r, \overrightarrow{\llbracket e \rrbracket_e}, \overrightarrow{x}, S'[i]\right) \end{pmatrix} \\
\\
\mathcal{S}^{\bullet}\left(P\right) &= P \\
\mathcal{S}^{ck \textbf{ on } C(x)}\left(P\right) &= \mathcal{S}^{ck}\left((x = C) \implies P\right)
\end{aligned}
$$

Fig. 9. State / transition semantics of Lustre

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 9 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 described in fig. 5. 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 $(a \implies b) \wedge (\neg a \implies c)$). 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 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.

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

$$f\_tr_i\left(S, \overrightarrow{I}, \overrightarrow{L_i}, \overrightarrow{O_i}, S'\right) \triangleq \exists \overrightarrow{V_i}, f\_tr_{i-1}\left(S, \overrightarrow{I}, \overrightarrow{L_{i-1}}, \overrightarrow{O_{i-1}}, S'\right) \wedge [\![eq_i]\!]_{eq}$$

$$f\_tr\left(S, \overrightarrow{I}, \overrightarrow{O}, S'\right) \triangleq f\_tr_n\left(S_r, \overrightarrow{I}, \overrightarrow{O}, S'\right)$$

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

4.2 C code specification: local annotations and function contracts

Eventually the logical annotations attached to Machine statements are translated into predicates, contracts and assertions in ACSL (ANSI/ISO C Specification Language), the specification language used by the Frama-C platform. 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 plugin that implements a weakest precondition calculus, contracts and assertions can be checked by external SMT solvers such as Alt-Ergo [24], CVC4 [4] or Z3 [25].

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 choose to 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. Such a structure is declared as *ghost*, that is an ACSL feature meaning that 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 $s_k = \mathcal{T}_{eq}(eq_k)$. 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

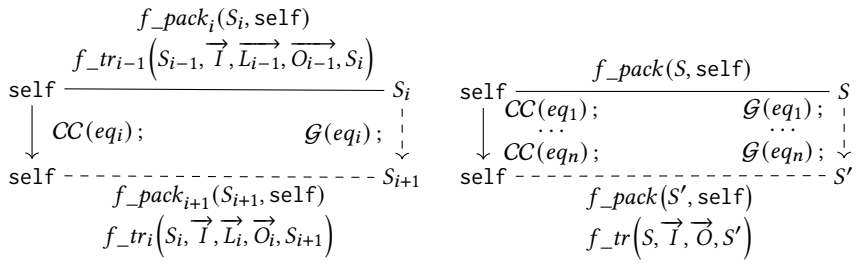


Fig. 10. Fine and coarse-grained simulation schemes

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, i.e. 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`.

$$f\_pack_k(S, \texttt{self}) \triangleq \bigwedge_{i \in \mathcal{I}_f} i\_pack(S[i], \texttt{self->i}) \wedge \bigwedge_{m \in \mathcal{S}_f} \neg\, arrow\_rst(S[Arrow(m)]_r) \implies S_k(m) = \texttt{self->m}$$

$$\text{Where } r = 0 \text{ if } Index[Arrow(m)] \leq k < Index(m), k \text{ otherwise}$$

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

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 $CC(eq_i)$, 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 10 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 $CC(eq_i)$ 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 and ghost state resetting. 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 with a special comment 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 for our example.

```
/*@ 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 that 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 and transition relations. 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 straightforwardly 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 that 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 CODE OPTIMIZATIONS AND IMPACT ON THE PROOF FRAMEWORK

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 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

```
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 }
  --@ clocks_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 } }
  --@ clocks_tr5(x, d, c, b2)
  case (d) { Up: case (c) { On: b1 := 1 } }
  --@ clocks_tr6(x, d, c, b1, b2)
  case (d) { Up: case (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

Fig. 11. Lustre example and non-optimized translated Machine code

the same definitions, the actual parameters $\overrightarrow{L_i}$ of $f\_tr_i(S, \overrightarrow{I}, \overrightarrow{L_i}, \overrightarrow{O_i}, S')$ may change according to the optimization level. Also, moving annotations around may yield capture problems. There are several ways of handling those issues, e.g. 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 that 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 on fig. 11a, that underlines the use of user-defined enumerated types as clocks. Figure 11b 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 12 presents the four optimizations on Machine code and figs. 13 and 14 details them on the example.

Conditionals fusion (cf. fig. 12a). 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 [8], implementation consists in merging adjacent conditional branches and regrouping their annotations. We can see on fig. 13a the high number of generated conditionals fused to produce better code.

Variable inlining (cf. fig. 12b). 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 kept alive in the specification. The transformed example code after fusion of conditionals and inlining of variables is presented

$$
\begin{array}{l}
k: \ \textbf{case}\ (e)\ \{\ \overrightarrow{C_i : s_{k,i}}\ \} \\
\quad \texttt{--@}\ f\_tr_k\left(\overrightarrow{I}, \overrightarrow{L_k}, \overrightarrow{O_k}\right) \\
k+1: \ \textbf{case}\ (e)\ \{\ \overrightarrow{C_i : s_{k+1,i}}\ \} \\
\quad \texttt{--@}\ f\_tr_{k+1}\left(\overrightarrow{I}, \overrightarrow{L_{k+1}}, \overrightarrow{O_{k+1}}\right)
\end{array}
\qquad
\begin{array}{l}
k: \ \textbf{case}\ (e)\ \{\ \overrightarrow{C_i : s_{k,i}; s_{k+1,i}}\ \} \\
\quad \texttt{--@}\ f\_tr_k\left(\overrightarrow{I}, \overrightarrow{L_k}, \overrightarrow{O_k}\right) \\
k+1: \ ; \\
\quad \texttt{--@}\ f\_tr_{k+1}\left(\overrightarrow{I}, \overrightarrow{L_{k+1}}, \overrightarrow{O_{k+1}}\right)
\end{array}
$$

(a) **Conditional fusion:** code is in SSA form.

$$
\begin{array}{l}
k: \ x := A; \\
\quad \texttt{--@}\ f\_tr_k\left(\overrightarrow{I}, \overrightarrow{L_k}, \overrightarrow{O_k}\right) \\
\quad \cdots \\
l: \ s_l \\
\quad \texttt{--@}\ f\_tr_l\left(\overrightarrow{I}, \overrightarrow{L_l}, \overrightarrow{O_l}\right)
\end{array}
\qquad
\begin{array}{l}
k: \ \texttt{--@}\ x := A \\
\quad \texttt{--@}\ f\_tr_k\left(\overrightarrow{I}, \overrightarrow{L_k}, \overrightarrow{O_k}\right) \\
\quad \cdots \\
l: \ [A/x]s_l \\
\quad \texttt{--@}\ f\_tr_l\left(\overrightarrow{I}, \overrightarrow{L_l}, \overrightarrow{O_l}\right)
\end{array}
$$

(b) **Variable inlining:** A is either a constant or a variable, code is in SSA form. Applied to all lines $l \geq k$.

$$
\begin{array}{l}
k: \ y := e; \\
\quad \texttt{--@}\ f\_tr_k\left(\overrightarrow{I}, \overrightarrow{L_k}, \overrightarrow{O_k}\right) \\
\quad \cdots \\
l: \ x := e'; \\
\quad \texttt{--@}\ f\_tr_l\left(\overrightarrow{I}, \overrightarrow{L_l}, \overrightarrow{O_l}\right) \\
\quad \cdots \\
m: \ s_m \\
\quad \texttt{--@}\ f\_tr_m\left(\overrightarrow{I}, \overrightarrow{L_m}, \overrightarrow{O_m}\right)
\end{array}
\qquad
\begin{array}{l}
k: \ y := e; \\
\quad \texttt{--@}\ y' := y \\
\quad \texttt{--@}\ f\_tr_k\left(\overrightarrow{I}, \overrightarrow{L_k}, \overrightarrow{O_k}\right) \\
\quad \cdots \\
l: \ y := e'; \\
\quad \texttt{--@}\ x := y \\
\quad \texttt{--@}\ f\_tr_l\left(\overrightarrow{I}, [y'/y]\overrightarrow{L_l}, \overrightarrow{O_l}\right) \\
\quad \cdots \\
m: \ [y/x]s_m \\
\quad \texttt{--@}\ f\_tr_m\left(\overrightarrow{I}, [y'/y]\overrightarrow{L_m}, \overrightarrow{O_m}\right)
\end{array}
$$

(c) **Variable recycling:** either $\mathtt{y} \notin \mathcal{L}_l$ or $\mathtt{y}$ is clock-disjoint from $\mathtt{x}$. Applied to all lines $m \geq l$.

$$
\begin{array}{l}
k: \ \textbf{case}\ (e)\ \{\ \cdots\ C_i : x := D_j\ \cdots\ \} \\
\quad \texttt{--@}\ f\_tr_k\left(\overrightarrow{I}, \overrightarrow{L_k}, \overrightarrow{O_k}\right) \\
k+1: \ \textbf{case}\ (x)\ \{\ \cdots\ D_j : s_{k+1,j}\ \cdots\ \} \\
\quad \texttt{--@}\ f\_tr_{k+1}\left(\overrightarrow{I}, \overrightarrow{L_{k+1}}, \overrightarrow{O_{k+1}}\right)
\end{array}
\qquad
\begin{array}{l}
k: \ \textbf{case}\ (e)\ \{ \\
\quad \cdots \\
\quad C_i :\texttt{--@}\ x := D_j \\
\quad \ s_{k+1,j} \\
\quad \cdots \\
\quad \} \\
\quad \texttt{--@}\ f\_tr_k\left(\overrightarrow{I}, \overrightarrow{L_k}, \overrightarrow{O_k}\right) \\
k+1: \ ; \\
\quad \texttt{--@}\ f\_tr_{k+1}\left(\overrightarrow{I}, \overrightarrow{L_{k+1}}, \overrightarrow{O_{k+1}}\right)
\end{array}
$$

(d) **Enumerated type elimination:** $x \notin \mathcal{L}_{k+1}$, code is in SSA form.

Fig. 12. The four optimizations: original code (left) vs. optimized code (right).

in fig. 13b. The variables `b1`, `b2`, `b3`, `c1` and `c2` are inlined in the statements but turned into ghost variables in the specification.

Variable recycling (cf. fig. 12c). 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 in order to reuse dead variables or clock-disjoint ones, i.e. 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 not in SSA (Static Single-Assignment) form anymore, 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

```
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;
      if (c2) { c := Off }
      else { c := On }
      case (c) {
        On:
          b1 := 1;
          z := b1
        Off:
          b2 := 2;
          z := b2
      }
      y := z;
    Down:
      b3 := 3;
      y := b3
  }
  --@ 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)
}
```

(a) Conditionals fusion

```
step(x: int) returns (y: int)
var c: en1; d: en2; z: int
{
  --@ c1 := x >= 0
  --@ clocks_tr1(x, c1)
  if (x >= 0) { d := Up }
  else { d := Down }
  --@ clocks_tr2(x, d)
  case (d) {
    Up:
      --@ c2 := x = 0
      if (x = 0) { c := Off }
      else { c := On }
      case (c) {
        On:
          --@ b1 := 1
          z := 1
        Off:
          --@ b2 := 2
          z := 2
      }
      y := z
    Down:
      --@ b3 := 3
      y := 3
  }
  --@ 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)
}
```

(b) + Variable inlining

Fig. 13. Optimizations effects on Machine code and annotations on example (I).

substituted for y. On the example in fig. 14a, 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 (cf. fig. 12d). For a variable x belonging in an enumerated type (e.g. 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 on the code in fig. 14b that the 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

In the remaining of the section, recall that LustreC optimization levels correspond to the following: O-1 is no optimization at all, O1 is conditionals fusion, O2 adds variable inlining and O3 adds variable recycling and enum elimination (see section 5).

```
step(x: int) returns (y: int)
var c: en1; d: en2
{
  --@ c1 := x >= 0
  --@ clocks_tr1(x, c1)
  if (x >= 0) { d := Up }
  else { d := Down }
  --@ clocks_tr2(x, d)
  case (d) {
    Up:
      --@ c2 := x = 0
      if (x = 0) { c := Off }
      else { c := On }
      case (c) {
        On:
          --@ b1 := 1
          y := 1
          --@ z := y
        Off:
          --@ b2 := 2
          y := 2
          --@ z := y
      }
    Down:
      --@ b3 := 3
      y := 3
  }
  --@ 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)
}
```

(a) + Variable recycling

```
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)
}
```

(b) + Enum elimination

Fig. 14. Optimizations effects on Machine code and annotations on example (II).

6.1 Translation validation

To evaluate our compiler extension, we ran it against a set of Lustre programs taken from the benchmarking suite of the Kind tool [29], which contains various applications, from microwave controllers to cache protocols, and from the test suite of the CoCoSim tool [10]. We used the default level of optimization of the compiler (O2), that is conditionals fusion and variable inlining (see fig. 13b). 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 15 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. Unsuccessful tests can be classified in two categories:

Failed tests where the solvers cannot manage to prove some obligations without manual guidance. This includes writing supplementary assertions in the code, finding manually a witness for

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

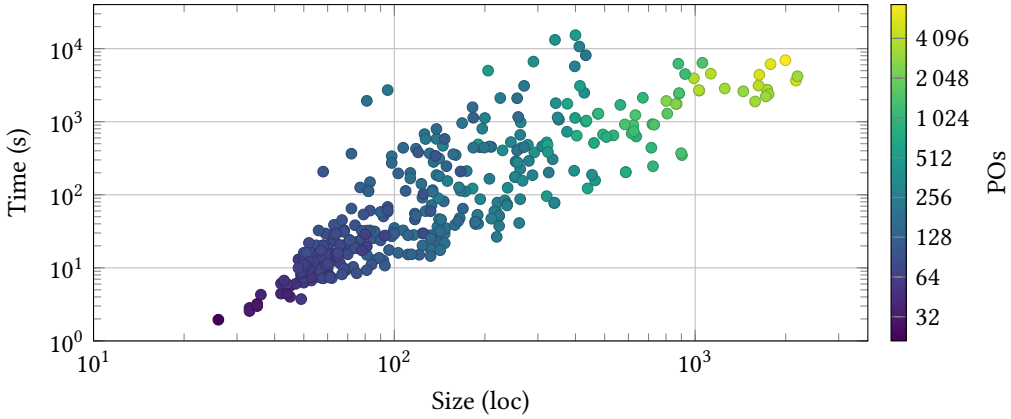


Fig. 15. Experiments report with O2 optimization level

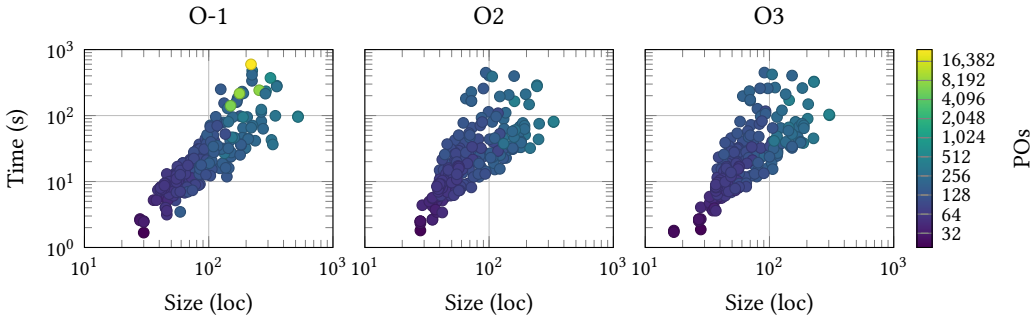


Fig. 16. Experiments report for the selected 220 test files

an existentially quantified variable, unfolding a predicate definition, etc. We developed a Frama-C *plugin* in OCaml to automatically perform some of these proof steps, but there are 23 cases (6%) for which it is not sufficient. For these tests, verification succeeds nonetheless on all but a few such problematic POs (99.32%).

Timed-out tests whose verification cannot end before the global timeout, because they yield too many or too complex POs. For these 6 cases (2%), it is difficult to predict whether increasing the timeout will lead to successful verification or not.

The O3 optimization level adds variable recycling and enumerated type elimination to optimizations already provided by the O2 level. In order to check its correctness and scalability with respect to the number of generated POs and time needed to prove them, we selected 220 test files from our initial set that were proved correct with the O2 level in less than 450 s and tested them with O3. First, notice that all O3 generated files are also proved. Figure 16 presents the same scattered log-log plots than fig. 15 for the selected 220 test files respectively for O-1, O2 and O3 levels. We added O-1

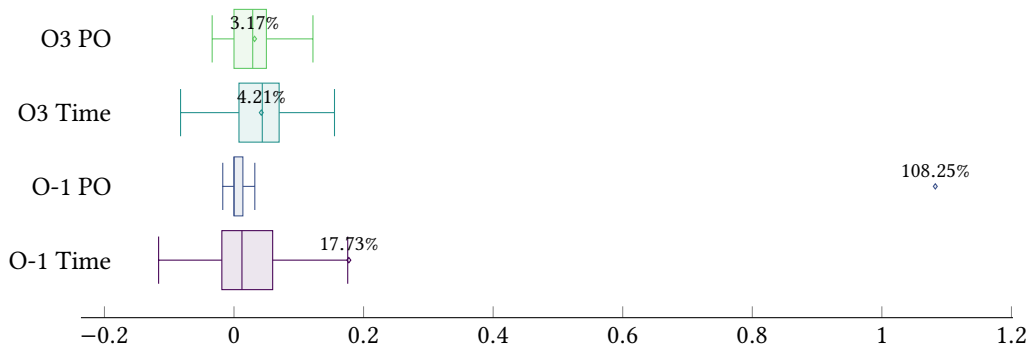


Fig. 17. Relative changes of number of POs and verification time relatively to O2

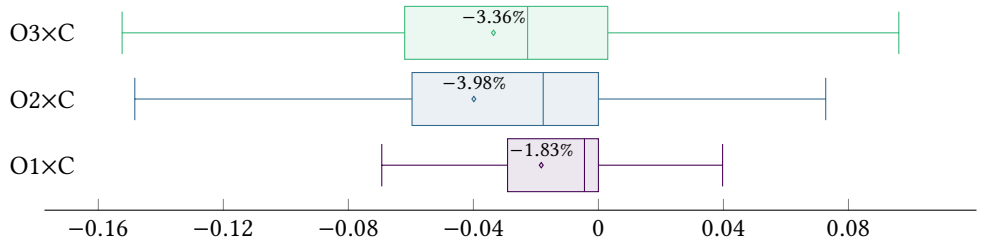


Fig. 18. Relative change of WCET relatively to O-1×C

for reference, even though the generated code is not realistic. Notice that the optimization level has no large impact on the characteristics of the generated code: both number of generated POs and timeouts are roughly linear with respect to the size of the generated C code. Figure 17 details more specifically the comparison between O-1, O2 and O3 optimization levels. We use box plots to display the relative changes of both number of POs and verification time ($(x_{Oi} - x_{O2})/x_{O2}$). Outliers are not shown. The average value in percentage is also displayed. As the average number of POs increases for O3 level, an increase of the verification time is expected. The remaining overhead is probably due to the fact that at O3 level, the respective structures of the generated code and the generated specification annotations do not match anymore. We observe an explosion of the number of POs and associated verification time in some cases with O-1, which is due to the higher number of conditionals.

6.2 Performance evaluation

To evaluate the performance gain of the generated code allowed by the optimizations, we provide experimental estimation of both worst-case execution time (WCET) and stack usage. In the remaining, we use the following notation to describe the different combinations that we evaluate: Oi×B, where O stands for LustreC optimization level, $i \in \{-1, 1, 2, 3\}$, and B, for Backend, is either C for CompCert or Gj for GCC with $j \in \{0, 1, 2, 3\}$ optimization level.

6.2.1 WCET estimation. To estimate the WCET, we follow the workflow of [11] and use the OTAWA v2 tool [3] with default settings. Figure 18 shows the effect of LustreC optimizations when compiling the generated code with CompCert. The O3 level targets particularly stack usage,

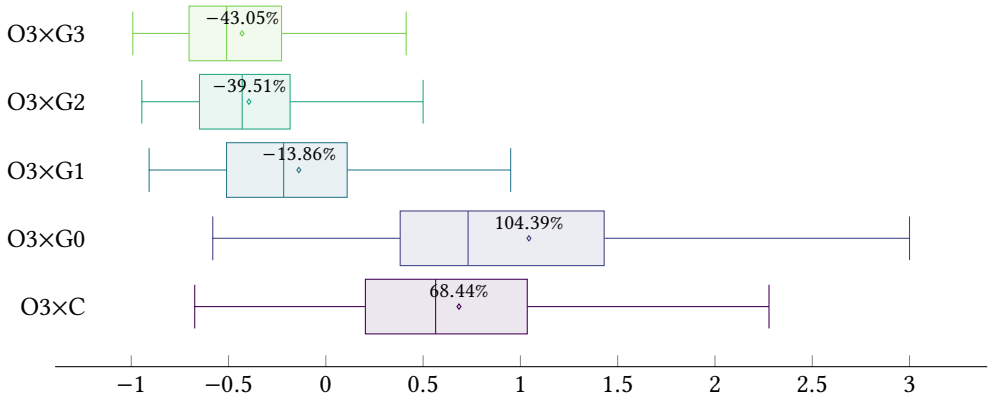


Fig. 19. Relative change of WCET relatively to Vélus

so the fact that it does not improve the WCET compared to O2 level is not surprising. These results show that when compiling with CompCert, which does perform some optimizations by default, implementing optimizations upstream in LustreC can be useful to improve the WCET of the executable code.

Figure 19 shows the comparison with Vélus. It appears that unless GCC is used with optimizations enabled, the code generated by Vélus is more efficient. It could be a surprising result, since the only optimization that Vélus implements is conditionals fusion. As outlined in [11] §5, the obtained results with OTAWA favor Vélus because contrary to other compilers like Heptagon [26], Lustre V6 [31] or LustreC, it does not compile the -> operator as a special node. Hence the code generated by Vélus saves a lot of extra function calls. However, in LustreC the `arrow_step` function is declared `inline` in order to mitigate this effect (except when using G0 which disables inlining). Therefore, the most important difference mainly originates in the design choices on the compilation of the modular reset. Vélus implements a "monolithic" recursive reset which inefficiently causes redundant resettings in some cases but saves the conditional statement at the beginning of each step function. Since almost none of the tests use the modular reset, the results strongly favor Vélus and we cannot evaluate this trade-off precisely. It would be possible to optimize away those conditional statements when it is statically certain that no reset will occur, but this is left as a future work. Note that not all our tests are successfully compiled by Vélus: in its current state, Vélus supports normalization [13] but not enumeration types and automata (yet) for example. Nonetheless, there are less than 10 such tests in our current benchmark.

Finally, fig. 20 shows that performing optimizations upstream in LustreC is not useful if compiling with an aggressively optimizing compiler such as GCC. Indeed, starting from G1 level, the generated code without any LustreC optimization will be (way) more efficient than the code generated with O2 level (O3 does not improve on the WCET, as seen above) but not optimized by GCC. Of course, the advantage of our proposal is that upstream optimizations are supported by the translation validation process, therefore they are validated if the verification succeeds. On the other hand, GCC optimizations are not validated, cannot be trusted and are therefore forbidden in the context of critical applications.

6.2.2 Stack usage estimation. Stack usage is another relevant measure in the embedded context. We estimate it on our benchmark only with compiling with GCC as a backend, as it provides

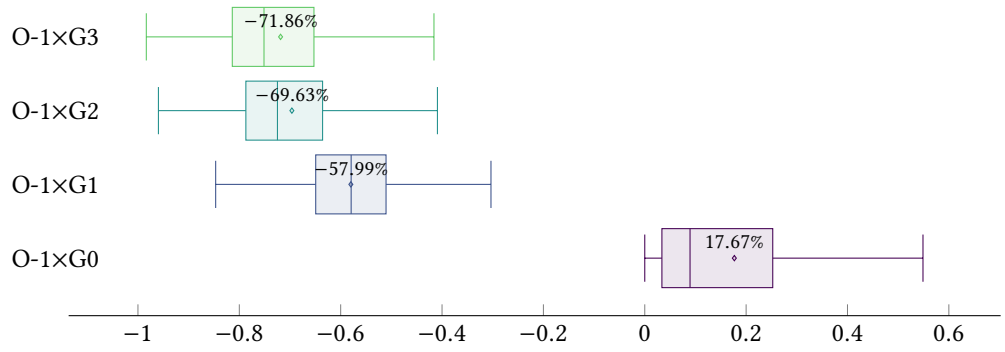


Fig. 20. Relative change of WCET relatively to O2×G0

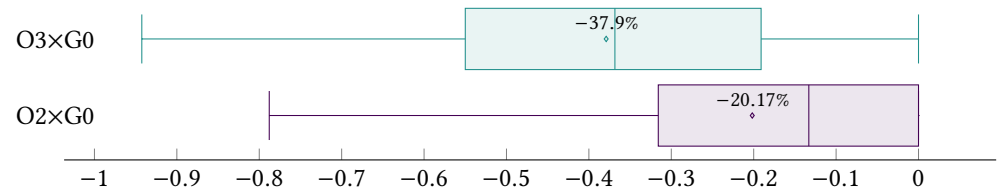


Fig. 21. Relative change of stack usage relatively to O-1×G0

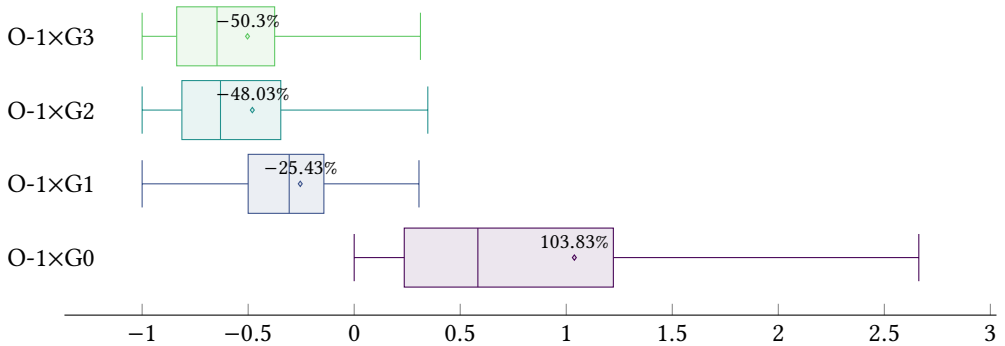


Fig. 22. Relative change of stack usage relatively to O3×G0

a dedicated reporting feature (through the flag `-fstack-usage`). There is existing work [15] to extend CompCert with a similar feature but we could not use it as it targets out-dated versions of CompCert and could not be easily interfaced with Vélus anyway.

Figure 21 shows the effect of LustreC optimizations on the stack usage, without any further optimizations by GCC. Contrary to the effect on the WCET, the gain of O3 level on stack usage is obvious, as those optimizations aim at reducing the number of variables in the generated code.

Finally, fig. 22 shows that once again, the upstream optimizations of LustreC are useless if GCC is used as a backend with aggressive optimizations enabled. But the gain is important if no GCC optimizations are performed. Indeed, when targetting modern processor architecture, GCC can

transfer most stack usage into registers, usually available in great numbers. Again, in the context of restricted processor architectures used in critical applications, GCC may not be able to do so.

7 CONCLUSION AND PERSPECTIVES

We succeeded in automatically providing to each Lustre source code an abstract operational semantics and proving with high success rates such a specification 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 including hierarchical state machines, 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 [41], 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.

ACKNOWLEDGMENTS

This work is supported by the Defense Innovation Agency (AID) of the French Ministry of Defense (research project CLEDESCHAMPS N 2021 65 0070).

REFERENCES

[1] Hafiz Muhammad Amjad, Kai Hu, Jianwei Niu, Noor Khan, Loïc Besnard, and Jean-Pierre Talpin. 2019. Translation Validation of Code Generation from the SIGNAL Data-Flow Language to Verilog. In *2019 15th International Conference on Semantics, Knowledge and Grids (SKG)* (SKG'19). (Sept. 18, 2019), 153–160. DOI: 10.1109/SKG49510.2019.00034.

[2] ANSYS. 2021. Scade suite. https://www.ansys.com/products/embedded-software/ansys-scade-suite.

[3] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. 2010. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In *Software Technologies for Embedded and Ubiquitous Systems* (Lecture Notes in Computer Science). Sang Lyul Min, Robert Pettit, Peter Puschner, and Theo Ungerer, (Eds.) Springer, Berlin, Heidelberg, (Oct. 15, 2010), 35–46. ISBN: 978-3-642-16256-5. DOI: 10.1007/978-3-642-16256-5_6.

[4] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings* (Lecture Notes in Computer Science). Ganesh Gopalakrishnan and Shaz Qadeer, (Eds.) Vol. 6806. Springer, (July 20, 2011), 171–177. DOI: 10.1007/978-3-642-22110-1_14.

[5] Patrick Baudin et al. 2021. The dogged pursuit of bug-free C programs: the Frama-C software analysis platform. *Communications of the ACM*, 64, 8, (July 26, 2021), 56–68. DOI: 10.1145/3470569.

[6] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. 2003. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91, 1, (Jan. 29, 2003), 64–83. DOI: 10.1109/JPROC.2002.805826.

[7] Albert Benveniste and Paul Le Guernic. 1990. Hybrid dynamical systems theory and the Signal language. *IEEE Transactions on Automatic Control*, 35, 5, (May 1990), 535–546. DOI: 10.1109/9.53519.

[8] Dariusz Biernacki, Jean-Louis Colaço, Gregoire Hamon, and Marc Pouzet. 2008. Clock-directed Modular Code Generation for Synchronous Data-flow Languages. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (LCTES '08). ACM, New York, NY, USA, (June 12, 2008), 121–130. ISBN: 978-1-60558-104-0. DOI: 10.1145/1375657.1375674.

[9] Brandon Bohrer, Yong Kiam Tan, Stefan Mitsch, Magnus O. Myreen, and André Platzer. 2018. VeriPhy: verified controller executables from verified cyber-physical system models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2018). Association for Computing Machinery, New York, NY, USA, (June 11, 2018), 617–630. ISBN: 978-1-4503-5698-5. DOI: 10.1145/3192366.3192406.

[10] Hamza Bourbouh, Pierre-Loïc Garoche, Thomas Loquen, Eric Noulard, and Claire Pagetti. 2020. CoCoSim, a Code Generation Framework for Control/Command Applications: an Overview of CoCoSim for Multi-Periodic Discrete Simulink Models. In *Embedded Real Time Systems (ERTS) 2020* (ERTS'20). (Jan. 29, 2020).

[11] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. 2017. A Formally Verified Compiler for Lustre. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI'17). ACM, Barcelona, Spain, (June 14, 2017), 586–601. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062358.

[12] Timothy Bourke, Lélio Brun, and Marc Pouzet. 2019. Mechanized semantics and verified compilation for a dataflow synchronous language with reset. In *Proceedings of the 47th ACM SIGPLAN Symposium on Principles of Programming Languages* (POPL'20). Vol. 4. ACM, New Orleans, LA, USA, (Dec. 20, 2019), 29. DOI: 10.1145/3371112.

[13] Timothy Bourke, Paul Jeanmaire, Basile Pesin, and Marc Pouzet. 2021. Verified Lustre Normalization with Node Subsampling. *ACM Transactions on Embedded Computing Systems*, 20, (Sept. 22, 2021), 98:1–98:25, 5s, (Sept. 22, 2021). DOI: 10.1145/3477041.

[14] Lélio Brun. 2020. *Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset*. Ph.D. Dissertation. École normale supérieure - PSL Research University, (July 6, 2020). 258 pp. Retrieved Mar. 18, 2021 from https://www.leliobrun.net/files/thesis.pdf.

[15] Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. 2014. End-to-end verification of stack-space bounds for C programs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '14). Association for Computing Machinery, New York, NY, USA, (June 9, 2014), 270–281. ISBN: 978-1-4503-2784-8. DOI: 10.1145/2594291.2594301.

[16] Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, Stavros Tripakis, and Peter Niebert. 2003. From Simulink to SCADE/Lustre to TTA: a Layered Approach for Distributed Embedded Applications. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems* (LCTES '03). ACM, New York, NY, USA, (June 11, 2003), 153–162. ISBN: 978-1-58113-647-0. DOI: 10.1145/780732.780754.

[17] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Alexander Plaice. 1987. LUSTRE: a declarative language for programming synchronous systems. In *In 14th Symposium on Principles of Programming Languages (POPL'87). ACM* (POPL'87). (Oct. 1, 1987). DOI: 10.1145/41625.41641.

[18] Ana Cavalcanti, Phil Clayton, and Colin O'Halloran. 2011. From control law diagrams to Ada via Circus. *Formal Aspects of Computing*, 23, 4, (July 1, 2011), 465–512. DOI: 10.1007/s00165-010-0170-3.

[19] Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli. 2016. The Kind 2 Model Checker. In *Proceedings of the 28th International Conference on Computer Aided Verification* (CAV'16). Swarat Chaudhuri and Azadeh Farzan, (Eds.) Vol. 9780. Springer, Toronto, ON, Canada, (July 13, 2016), 510–517. DOI: 10.1007/978-3-319-41540-6_29.

[20] Van Chan Ngo, Jean-Pierre Talpin, and Thierry Gautier. 2015. Translation Validation for Synchronous Data-Flow Specification in the SIGNAL Compiler. In *Formal Techniques for Distributed Objects, Components, and Systems* (Lecture Notes in Computer Science). Susanne Graf and Mahesh Viswanathan, (Eds.) Springer International Publishing, Cham, (June 4, 2015), 66–80. ISBN: 978-3-319-19195-9. DOI: 10.1007/978-3-319-19195-9_5.

[21] Mingshuai Chen, Xiao Han, Tao Tang, Shuling Wang, Mengfei Yang, Naijun Zhan, Hengjun Zhao, and Liang Zou. 2017. MARS: a Toolchain for Modelling, Analysis and Verification of Hybrid Systems. In *Provably Correct Systems*. NASA Monographs in Systems and Software Engineering. Mike Hinchey, Jonathan P. Bowen, and Ernst-Rüdiger Olderog, (Eds.) Springer International Publishing, Cham, (Mar. 2, 2017), 39–58. ISBN: 978-3-319-48628-4. DOI: 10.1007/978-3-319-48628-4_3.

[22] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. 2017. SCADE 6: a formal language for embedded critical software development. In *2017 International Symposium on Theoretical Aspects of Software Engineering (TASE)* (TASE'17). (Sept. 15, 2017), 1–11. DOI: 10.1109/TASE.2017.8285623.

[23] Jean-Louis Colaço and Marc Pouzet. 2003. Clocks as First Class Abstract Types. In *Embedded Software* (Lecture Notes in Computer Science). Springer, Berlin, Heidelberg, (Oct. 15, 2003), 134–155. ISBN: 978-3-540-45212-6. DOI: 10.1007/978-3-540-45212-6_10.

[24] Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala, and Alain Mebsout. 2018. Alt-Ergo 2.2. In *SMT Workshop: International Workshop on Satisfiability modulo Theories*. Oxford, United Kingdom, (July 13, 2018).

[25] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (TACAS'08/ETAPS'08). Springer-Verlag, Berlin, Heidelberg, (Mar. 29, 2008), 337–340. ISBN: 978-3-540-78799-0. DOI: 10.1007/978-3-540-78800-3_24.

[26] Gwenaël Delaval, Adrien Guatto, Hervé Marchand, Marc Pouzet, and Rutten Éric. 2017. *Heptagon/BZR manual*. PARKAS (ENS) and Ctrl-A (LIG/Inria). (Apr. 3, 2017). https://gitlab.inria.fr/synchrone/heptagon/-/blob/master/manual/heptagon-manual.pdf.

[27] Bernd Finkbeiner, Stefan Oswald, Noemi Passing, and Maximilian Schwenger. 2020. Verified Rust Monitors for Lola Specifications. In *Runtime Verification: 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6–9, 2020, Proceedings* (RV'20). Springer-Verlag, Berlin, Heidelberg, (Oct. 2, 2020), 431–450. ISBN: 978-3-030-60507-0. DOI: 10.1007/978-3-030-60508-7_24.

[28] Pierre-Loïc Garoche, Arie Gurfinkel, and Temesghen Kahsai. 2014. Synthesizing modular invariants for synchronous code. In *Proceedings First Workshop on Horn Clauses for Verification and Synthesis, HCVS 2014, Vienna, Austria, 17 July 2014* (EPTCS). Nikolaj S. Bjørner, Fabio Fioravanti, Andrey Rybalchenko, and Valerio Senni, (Eds.) Vol. 169. (Dec. 3, 2014), 19–30. DOI: 10.4204/EPTCS.169.4.

[29] George Hagen and Cesare Tinelli. 2008. Scaling up the formal verification of Lustre programs with SMT-based techniques. In *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design* (FMCAD '08). IEEE Press, Portland, Oregon, (Nov. 17, 2008), 1–9. ISBN: 978-1-4244-2735-2. DOI: 10.1109/FMCAD.2008.ECP.19.

[30] Grégoire Hamon and Marc Pouzet. 2000. Modular Resetting of Synchronous Data-flow Programs. In *Proceedings of the 2Nd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (PPDP '00). ACM, New York, NY, USA, (Sept. 1, 2000), 289–300. ISBN: 978-1-58113-265-6. DOI: 10.1145/351268.351300.

[31] Erwan Jahier, Pascal Raymond, and Nicolas Halbwachs. 2020. *The Lustre V6 Reference Manual*. Version 17-08-20. Verimag. http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v6/doc/lv6-ref-man.pdf.

[32] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52, 7, (July 1, 2009), 107–115. DOI: 10.1145/1538788.1538814.

[33] Mathworks. 2021. Simulink. https://www.mathworks.com/products/simulink.

[34] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems* (Lecture Notes in Computer Science). Bernhard Steffen, (Ed.) Springer Berlin Heidelberg, (Mar. 28, 1998), 151–166. ISBN: 978-3-540-69753-4. DOI: 10.1007/BFb0054170.

[35] Amir Pnueli, Ofer Strichman, and Michael Siegel. 1998. Translation Validation for Synchronous Languages. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming* (ICALP '98). Springer-Verlag, Berlin, Heidelberg, (July 17, 1998), 235–246. ISBN: 978-3-540-64781-2. DOI: 10.1007/BFb0055057.

[36] Amir Pnueli, Ofer Strichman, and Michael Siegel. 2000. Translation Validation: from SIGNAL to C. In *Correct System Design: Recent Insights and Advances*. Lecture Notes in Computer Science. Ernst-Rüdiger Olderog and Bernhard Steffen, (Eds.) Springer Berlin Heidelberg, Berlin, Heidelberg, (Mar. 24, 2000), 231–255. ISBN: 978-3-540-48092-1. DOI: 10.1007/3-540-48092-7_11.

[37] Michael Ryabtsev and Ofer Strichman. 2009. Translation Validation: from Simulink to C. In *Computer Aided Verification* (Lecture Notes in Computer Science). Ahmed Bouajjani and Oded Maler, (Eds.) Springer, Berlin, Heidelberg, (July 2, 2009), 696–701. ISBN: 978-3-642-02658-4. DOI: 10.1007/978-3-642-02658-4_57.

[38] Norman R. Scaife, Christos Sofronis, Paul Caspi, Stavros Tripakis, and Florence Maraninchi. 2004. Defining and Translating a "Safe" Subset of Simulink/Stateflow into Lustre. In *Proceedings of the 4th ACM International Conference on Embedded Software* (EMSOFT '04). ACM, New York, NY, USA, (Sept. 27, 2004), 259–268. ISBN: 978-1-58113-860-3. DOI: 10.1145/1017753.1017795.

[39] Gang Shi, Yuanke Gan, Shu Shang, Shengyuan Wang, Yuan Dong, and Pen-Chung Yew. 2017. A Formally Verified Sequentializer for Lustre-like Concurrent Synchronous Data-flow Programs. In *Proceedings of the 39th International Conference on Software Engineering Companion* (ICSE-C '17). IEEE Press, Piscataway, NJ, USA, (May 28, 2017), 109–111. ISBN: 978-1-5386-1589-8. DOI: 10.1109/ICSE-C.2017.83.

[40] Gang Shi, Yucheng Zhang, Shu Shang, Shengyuan Wang, Yuan Dong, and Pen-Chung Yew. 2019. A formally verified transformation to unify multiple nested clocks for a Lustre-like language. *Science China Information Sciences*, 62, 1, (Jan. 2019), 12801. DOI: 10.1007/s11432-016-9270-0.

[41] The Coq Team. 2021. Coq. https://coq.inria.fr.

[42] Xavier Thirioux and Pierre-Loïc Garoche. 2021. Lustrec. https://github.com/Embedded-SW-VnV/lustrec.

[43] Andres Toom, Nassima Izerrouken, Tõnu Näks, Marc Pantel, and Olivier Ssi Yan Kai. 2010. Towards Reliable Code Generation with an Open Tool: evolutions of the Gene-Auto toolset. In *ERTS2 2010, Embedded Real Time Software & Systems*. Toulouse, France, (May 19, 2010).

[44] Andres Toom, Tõnu Näks, Marc Pantel, Marcel Gandriau, and I. Wati. 2008. Gene-Auto: an Automatic Code Generator for a safe subset of Simulink/Stateflow and Scicos. In *Embedded Real Time Software and Systems (ERTS2008)*. Toulouse, France, (Feb. 1, 2008).

[45] Stavros Tripakis, Christos Sofronis, Paul Caspi, and Adrian Curic. 2005. Translating discrete-time simulink to lustre. *ACM Transactions on Embedded Computing Systems*, 4, 4, (Nov. 1, 2005), 779–818. DOI: 10.1145/1113830.1113834.

[46] Zhibin Yang, Jean-Paul Bodeveix, Mamoun Filali, Kai Hu, Yongwang Zhao, and Dianfu Ma. 2016. Towards a verified compiler prototype for the synchronous language SIGNAL. *Frontiers of Computer Science*, 10, 1, (Feb. 2016), 37–53. DOI: 10.1007/s11704-015-4364-y.

[47] Liang Zou, Naijun Zhan, Shuling Wang, and Martin Fränzle. 2015. Formal Verification of Simulink/Stateflow Diagrams. In *Automated Technology for Verification and Analysis* (Lecture Notes in Computer Science). Bernd Finkbeiner, Geguang Pu, and Lijun Zhang, (Eds.) Springer International Publishing, Cham, (Nov. 22, 2015), 464–481. ISBN: 978-3-319-24953-7. DOI: 10.1007/978-3-319-24953-7_33.

[48] Liang Zou, Naijun Zhan, Shuling Wang, Martin Fränzle, and Shengchao Qin. 2013. Verifying Simulink Diagrams via a Hybrid Hoare Logic Prover. In *Proceedings of the Eleventh ACM International Conference on Embedded Software* (EMSOFT '13). IEEE Press, Piscataway, NJ, USA, (Sept. 29, 2013), 9:1–9:10. ISBN: 978-1-4799-1443-2. DOI: 10.1109/EMSOFT.2013.6658587.