SCIENCE CHINA
Information Sciences

• HIGHLIGHT •

January 2019, Vol. 62 012801:1–012801:3
https://doi.org/10.1007/s11432-016-9270-0

A formally verified transformation to unify multiple nested clocks for a Lustre-like language

Gang SHI[1*], Yucheng ZHANG[1], Shu SHANG[1], Shengyuan WANG[1*], Yuan DONG[1] & Pen-Chung YEW[2*]

[1]*Department of Computer Science and Technology, Tsinghua University, Beijing* 100084*, China;*
[2]*Department of Computer Science and Engineering, University of Minnesota, Minneapolis* 55455*, USA*

Received 29 June 2017/Revised 31 August 2017/Accepted 6 September 2017/Published online 27 June 2018

Citation Shi G, Zhang Y C, Shang S, et al. A formally verified transformation to unify multiple nested clocks for a Lustre-like language. Sci China Inf Sci, 2019, 62(1): 012801, https://doi.org/10.1007/s11432-016-9270-0

Multiple nested clocks is a major language feature in synchronous data-flow languages such as Lustre [1]. To build a formally verified compiler for such a language, it is a common practice to compile the source program to a C-like program first before compiling it to low-level machine-dependent code using a formally verified backend compiler such as the CompCert compiler [2, 3]. Using this approach, it is necessary to convert multiple nested clocks to a single unified clock, a process called clock unification. This process can be done before or after the translation of the temporal operators such as `pre` and `fby`. This article gives a formal verification framework of the clock unification performed before the translation of temporal operators. We present such a program transformation using a small subset of Lustre V6 [4], called SL.

The SL language. The syntax of SL is shown below in a BNF grammar, in which the bold words, such as node, var and returns, are reserved words. An LS program consists of nodes defined by fd. Each node can have two types of equations in its body, defined by eq. Most of the operators in expr are similar to those in C language except that they operate on streams. A stream can be seen as an infinite or finite series of values taken in each basic clock cycle. In each clock cycle i, a stream s either takes a single value u, in which case we write $s[i] = u$; or does not take any value, in which case we write $s[i] = \perp$. Thus a stream can be seen as a partial function from $\mathbb{N}$ (the set of non-negative integers) to the domain of semantic values.

τ	::=	**int** | **bool**
prog	::=	(fd$^+$,MainId)
fd	::=	**node** id(args) **returns** (rets) body
body	::=	**var** vars **let** eq* **tel**
args,vars, rets	::=	(id, τ)*
eq	::=	id = expr
	|	id = id$_1$(expr*)
expr	::=	const | id
	|	uop expr$_1$ | expr$_1$ bop expr$_2$
	|	**if** (expr$_1$, expr$_2$, expr$_3$)
	|	**fby** (expr$_1$,expr$_2$)
	|	expr **when** id
	|	**current** (expr$_1$,expr$_2$, init)
n	::=	Nat(natural numbers)
init	::=	const
const	::=	n | **true** | **false**

Each stream is characterized by a type and a clock. All values in a stream must be of the same type. The clock c of a stream s determines whether s should have a value in a basic clock cycle or not. The basic clock is a reference clock, which is determined when a node is called. In this article, a clock can be defined as clock ::= **basic** | id, where **basic** denotes the basic clock, and id is a boolean variable identifier. For clarity, in the rest of this article, we often use $\widetilde{x}$ as the clock denoted by the boolean variable x. We use clock(e) to denote the clock of the expression e, which can be obtained from the static analyzer based on a type system

* Corresponding author (email: sg11@mails.tsinghua.edu.cn, wwssyy@tsinghua.edu.cn, Yew@cs.umn.edu)

for the clock checking (or clock calculus).

Clocks can be nested, that is, if the clock of x is sampled by x', the clock of x' in turn could be sampled by another variable x''. We can have clocks nested arbitrarily, so long as the relation of sampling does not form a cycle.

Clock operators and temporal operators are key to the clock unification. We consider three primitives: `fby`, `when` and `current`. The clock operator `fby` provides the delay mechanism to the system; while the clock operators `when` and `current` changes the clock of an expression.

We assume all of the Lustre-like programs satisfy well formed properties, among which the following is the most important: (1) All of the variables appearing on the left-hand side of the equations in a node are distinct. (2) There exists no cycle in the node call graph. (3) There exists no cycle in the program dependence graph of a node.

To focus on the clock unifying algorithm, as well as in the implementation of a prototype in Coq[1)], the transformation of an equation like "id $=$ id$_1(e_1, e_2, \ldots, e_n)$" is separately considered and not included in the formulation of this article. Therefore, in the rest of the article, we assume that a program has only one single node.

Theorem 1. The well-formedness property for any SL program P is decidable.

Clock-unifying transformation. Our transformation can accept any statically-proved well-formed SL program P as input, and produce a new well-formed program Q in the clock-restricted subset of SL. This language subset restricts the use of clock operators, `when` and `current`, and only the basic clock of a node is allowed for all variables and expressions.

In the clock unification transformation, all variables in P are kept in Q, except that their clocks are converted to the basic clocks. For each equation $x = e$ in P, we construct in Q a new equation $x = \mathcal{Z}(e)$. To define $\mathcal{Z}$, we first define three auxiliary meta-functions: $\mathcal{B}$, $\mathcal{F}$ and $\mathcal{R}$. New IDs are assigned whenever these functions are applied. Corresponding to each new ID, a new equation is produced in Q.

$\mathcal{B}(c)$ takes a clock c in P as an input argument, and returns it as a boolean variable based on c in Q, that is, we have $\mathcal{B}(\texttt{basic}) = x'$, where x' is a variable defined as $x' = \texttt{true}$, and $\mathcal{B}(\widetilde{x}) = x'$, where x' is a variable defined as $x' = \texttt{if}(\mathcal{B}(\text{clock}(x)), x, \text{false})$.

$\mathcal{F}(x)$ takes a boolean variable x as the input argument, and returns a variable to detect the first cycle that x turns `true`. We define $\mathcal{F}(x) = x'$, where $x' = \texttt{fby}(\texttt{true}, \texttt{if}(x, \texttt{false}, x'))$.

$\mathcal{R}(e, x, u)$ takes as input an expression e, a boolean variable x, and a constant value u that has the same type as e. It returns a variable with the same type as e. If x is `true` in a cycle (i.e., it is sampled), $\mathcal{R}(e, x, u)$ assumes the same value as e in that cycle. If x is `false` (i.e., not sampled), $\mathcal{R}(e, x, u)$ returns the value of e in the cycle of which x was most recently sampled, i.e., in the most recent cycle in which x is `true`, or u if x has yet to be `true`. We have $\mathcal{R}(e, x, u) = x'$, where $x' = \texttt{if}(x, e, \texttt{fby}(u, x'))$.

The transformation function $\mathcal{Z}$ is then defined inductively based on the construction of the expression e, where we use u_0 to represent an arbitrary semantic value of the appropriate type in the specified context.

- $\mathcal{Z}(u) = u$, where u is a const expression.
- $\mathcal{Z}(x) = \mathcal{R}(x, \mathcal{B}(\text{clock}(x)), u_0)$.
- $\mathcal{Z}(\text{uop } e_1) = \text{uop } \mathcal{Z}(e_1)$
- $\mathcal{Z}(e_1 \text{ bop } e_2) = \mathcal{Z}(e_1) \text{ bop } \mathcal{Z}(e_2)$
- $\mathcal{Z}(\text{if}(e_1, e_2, e_3)) = \text{if}(\mathcal{Z}(e_1), \mathcal{Z}(e_2), \mathcal{Z}(e_3))$
- $\mathcal{Z}(e \texttt{ when } x) = \mathcal{R}(\mathcal{Z}(e), \mathcal{B}(\widetilde{x}), u_0)$.
- $\mathcal{Z}(\texttt{current}(e, u)) = \mathcal{R}(\mathcal{Z}(e), \mathcal{B}(\widetilde{x}), u)$.
- $\mathcal{Z}(\texttt{fby}(e_1, e_2)) =$ $\mathcal{R}(\texttt{if}(\mathcal{F}(\mathcal{B}(c)), \mathcal{Z}(e_1), \texttt{fby}(u_0, \mathcal{Z}(e_2))), \mathcal{B}(c), u_0)$,

where $c = \text{clock}(e_1)$.

Theorem 2. For any well-formed program P, the presented algorithm terminates and produces a well-formed program program Q, which is denoted as $P \rightrightarrows Q$.

Dynamic semantics. A program is formally defined as a tuple $P = (K, T, C, E)$. The map $K : \mathbb{N}^+ \to \texttt{kind}$ declares the kind of each variable, representing input, output and local variable, respectively. The map $T : \mathbb{N}^+ \to \texttt{type}$ declares the type of each variable. The map $C : \mathbb{N}^+ \to \texttt{clock}$ declares the clock of each variable. The map $E : \mathbb{N}^+ \to \texttt{expr}$ defines each non-input variable x by an expression e, representing the equation $x = e$ in the program. Notice that we have assumed that a program only has one single node.

Variable identifiers in P are set to be in the set of positive integers $\mathbb{N}^+$ to simplify work in Coq. Moreover, for a finite list l, we use $l[i]$ to extract the ith element from the list.

The dynamic semantics is defined by inductively defining predicates of the form $s[i] = v$, indicating the stream s takes value v from the domain $\texttt{val}'$ in the cycle i of the basic clock. $\texttt{val}'$ is the lifted version of the semantic domain `val`, which is the domain of `val` plus $\perp$. The lifted version of `val` u in the domain $\texttt{val}'$ is denoted as $\lfloor u \rfloor$.

We introduce $I, O : \mathbb{N}^+ \to \texttt{list val}'$ to give each input or output variable a finite list of $\texttt{val}'$s, respectively. I and O will be used as components

1) The Coq Proof Assistant. https://coq.inria.fr/.

of the semantic environment.

$$\frac{I(x) = s \quad s[i] = v}{E, I \vdash x[i] \to v} \qquad \frac{E, I \vdash e_1[i] \to \lfloor u_0 \rfloor \quad E, I \vdash e_2[\curvearrowleft i] \overset{u_0}{\rightsquigarrow} u}{E, I \vdash \texttt{fby}(e_1, e_2)[i] \to \lfloor u \rfloor}$$

$$\frac{}{E, I \vdash u[i] \to \lfloor u \rfloor} \qquad \frac{E, I \vdash e_1[i] \to \bot}{E, I \vdash \texttt{fby}(e_1, e_2)[i] \to \bot}$$

$$\frac{E, I \vdash e[i] \to \lfloor u \rfloor \quad E, I \vdash x[i] \to \lfloor \texttt{true} \rfloor}{E, I \vdash (e \texttt{ when } x)[i] \to \lfloor u \rfloor} \qquad \frac{E, I \vdash e_1[i] \to \lfloor u_1 \rfloor \quad E, I \vdash e_2[i] \to \lfloor u_2 \rfloor \quad \underline{\text{bop}}(u_1, u_2) \to u}{E, I \vdash u_1 \text{ bop } u_2[i] \to \lfloor u \rfloor}$$

$$\frac{E, I \vdash e[i] \to \lfloor u \rfloor \quad E, I \vdash x[i] \to \lfloor \texttt{false} \rfloor}{E, I \vdash (e \texttt{ when } x)[i] \to \bot} \qquad \frac{E, I \vdash e_1[i] \to \bot \quad E, I \vdash e_2[i] \to \bot}{E, I \vdash u_1 \text{ bop } u_2[i] \to \bot}$$

$$\frac{E, I \vdash e[i] \to \bot \quad E, I \vdash x[i] \to \bot}{E, I \vdash (e \texttt{ when } x)[i] \to \bot} \qquad \frac{I(x) = s \quad s[i] = v}{E, I \vdash x[i] \to v}$$

$$\frac{E, I \vdash e[i] \to \lfloor u \rfloor \quad E, I \vdash x[i] \to \lfloor \texttt{true} \rfloor}{E, I \vdash \texttt{current}(e, x, u_0)[i] \to \lfloor u \rfloor} \qquad \frac{E(x) = e \quad E, I \vdash e[i] \to v}{E, I \vdash x[i] \to v}$$

$$\frac{}{E, I \vdash e[\curvearrowleft 0] \overset{u_0}{\rightsquigarrow} u_0}$$

$$\frac{E, I \vdash e[\curvearrowleft i] \overset{u_0}{\rightsquigarrow} u \quad E, I \vdash x[i] \to \lfloor \texttt{false} \rfloor}{E, I \vdash \texttt{current}(e, x, u_0)[i] \to \lfloor u \rfloor} \qquad \frac{E, I \vdash e[i] \to \lfloor u \rfloor}{E, I \vdash e[\curvearrowleft i+1] \overset{u_0}{\rightsquigarrow} u}$$

$$\frac{E, I \vdash e[i] \to \bot \quad E, I \vdash x[i] \to \bot}{E, I \vdash \texttt{current}(e, x, u_0)[i] \to \bot} \qquad \frac{E, I \vdash e[i] \to \bot \quad E, I \vdash e[\curvearrowleft i] \overset{u_0}{\rightsquigarrow} u}{E, I \vdash e[\curvearrowleft i+1] \overset{u_0}{\rightsquigarrow} u}$$

The dynamic semantics are defined by inference rules given above, in which the inductive judgements as follows are used.

$E, I \vdash e[i] \to v$ shows the expression e taking v in the ith basic clock cycle.

$E, I \vdash x[i] \to v$ shows the variable x taking v in the ith basic clock cycle.

$E, I \vdash e[\curvearrowleft i] \overset{u_0}{\rightsquigarrow} u$ shows the expression e taking u as the most recently defined value before the ith basic clock cycle. If there is no such defined value, u_0 is used instead. Equipped with this judgement, the semantics of `fby` and `current` can be described formally.

For the semantics of expressions, we only give the example of binary operators. We use $\underline{\text{bop}}(u_1, u_2) \to u$ to specify the semantic evaluation of the binary expression u_1 bop u_2.

The semantics of the whole program P is specified by the predicate $P_n(I) \to O$, which states that the first n basic clock cycles of P would produce output O from the input I. When P is well-formed, $P_n(I) \to O$ is expected to be a mapping of any valid input I to a valid output O.

For convenience, we use $I \sim P_n$ to denote the validity of an input I of program P on n basic clock cycles. It just states that the input values conform to the expected type and clock.

We omit the formal definitions for $P_n(I) \to O$ and $I \sim P_n$, as well as the proof of the soundness of the dynamic semantics defined above.

Semantics preservation. To formally prove that the transformation preserves the semantics, we first give a formal definition of refinement for an input or output.

We call an input I_2 a refinement of I_1 with respect to the types given by T, which we denote as $I_1 \sqsubseteq_T I_2$, if (1) I_1 and I_2 have the same input variable set; (2) $\forall x, s_1, s_2, I_1(x) = s_1 \wedge I_2(x) = s_2 \Rightarrow s_1$ and s_2 have the same length; (3) $\forall x, s_1, s_2, i, u, I_1(x) = s_1 \wedge I_2(x) = s_2 \wedge s_1[i] = \lfloor u \rfloor \Rightarrow u$ is of type t; (4) $\forall x, s, v, i, I_2(x) = s \wedge s[i] = v \Rightarrow \exists u, v = \lfloor u \rfloor$. Similarly, we can define $O_1 \sqsubseteq_{T_P} O_2$.

The refinement will keep the validity of input, as stated by the following theorem.

Theorem 3. For any well-formed program P and any program Q satisfying $P \rightrightarrows Q$, if $I_1 \sqsubseteq_{T_P} I_2$ where $I_1 \sim P_n$, then we have $I_2 \sim Q_n$.

Finally, the semantics preservation is basically established by the following theorem.

Theorem 4. For any well-formed program P and any program Q satisfying $P \rightrightarrows Q$, if $P_n(I_1) \to O_1$ and $Q_n(I_2) \to O_2$, where $I_1 \sim P_n$, then $I_1 \sqsubseteq_{T_P} I_2$ implies $O_1 \sqsubseteq_{T_P} O_2$.

Conclusion. A clock-unifying transformation of Lustre-like programs is proposed, and its semantics preservation is proved, which have been formulated and verified using Coq[2]. The `fby` operator in this article has two operands in accordance with Lustre V6. Therefore, the clock unifying algorithm in this article is not suitable for those languages with a three-operand `fby` operator, e.g., Scade[3].

Acknowledgements This work was supported by National Natural Science Foundation of China (Grant Nos. 61272086, 61462086, MJ-2015-D-066) and Sino-European Laboratory of Informatics, Automation and Applied Mathematics (Grants for the project Formally Certified Software Tools).

References

1 Halbwachs N, Caspi P, Raymond P, et al. The synchronous data flow programming language LUSTRE. Proc IEEE, 1991, 79: 1305–1320
2 Leroy X. Formal verification of a realistic compiler. Commun ACM, 2009, 52: 107–115
3 Bourke T, Brun L, Dagand P, et al. A formally verified compiler for Lustre. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, Barcelona, 2017. 586–601
4 Jahier E, Raymond P, Haibwachs N. The Lustre V6 Reference Manual. Software Version: master.737 (27-04-18), 2018

2) Source code to unify nested clocks in a lustre-like language. https://github.com/yczhang89/unify-clock.
3) ANSYS SCADE Suite. http://www.esterel-technologies.com/products/scade-suite.