Security Types for Synchronous Data Flow Systems

Sanjiva Prasad
IIT Delhi, India
sanjiva@cse.iitd.ac.in

R.Madhukar Yerraguntla
IIT Delhi, India
madhukar.yr@cse.iitd.ac.in

Subodh Sharma
IIT Delhi, India
svs@cse.iitd.ac.in

***Abstract*—Synchronous reactive data flow is a paradigm that provides a high-level abstract programming model for embedded and cyber-physical systems, including the locally synchronous components of IoT systems. Security in such systems is severely compromised due to low-level programming, ill-defined interfaces and inattention to security classification of data. By incorporating a Denning-style lattice-based secure information flow framework into a synchronous reactive data flow language, we provide a framework in which correct-and-secure-by-construction implementations for such systems can be specified and derived. In particular, we propose an extension of the LUSTRE programming framework with a security type system. We prove the soundness of our type system with respect to the co-inductive operational semantics of LUSTRE by showing that well-typed programs exhibit non-interference.**

***Index Terms*—: Synchronous reactive data flow, Lustre, Security lattice, Security type system, Stream semantics, Non-interference**

I. INTRODUCTION

The widespread deployment of millions of embedded and cyber-physical systems, especially in the Internet of Things paradigm, poses interesting challenges such as efficiency, scale, and, most significantly, of correctness of operation and *security* [30]. Numerous high-profile attacks such as those on CAN systems [27], [12], smart lighting [23], and pacemakers [17], [40] have exposed vulnerabilities in critical systems. These attacks exploit lacunae such as: (L1) ill-defined interfaces; (L2) no secure information flow (SIF) architecture and weak security mechanisms; (L3) components operating with greater privilege or more capability than necessary.

Our contention is that much of this unfortunate insecurity can be avoided by using a high-level programming paradigm. While Domain-Specific Languages suggest a principled way to build more reliable systems, we contend, in the tradition of Landin [35], that for the large subset of locally synchronous systems there already is a quintessential solution – namely, reactive synchronous data-flow languages.

There are several merits to adopting this paradigm: (M1) "Things", embedded and cyber-physical systems, can be abstractly treated as sources, sinks and transducers over (clocked) streams of data. Indeed, this extensional view supports not only abstract "things" but composite computations as first-class entities. (M2) The data flow model makes explicit all interfaces, connections and data dependencies, and (clocked, named) flows – thus greatly reducing attack surfaces and unanticipated interactions. (M3) In synchronous reactive data flow languages, monitoring safety properties is easy and can be achieved using finite-state automata [25]. Not only can monitors be expressed within the model, but the same framework can be used to specify axioms and assumptions, to constrain behaviour, and specify test cases [48]. (M4) LUSTRE [11], [24] is an eminently suitable synchronous reactive data flow language, for which there already exist elegant formal semantics *and* a suite of tools for (a) certified compilation from the high-level model into lower-level imperative languages [7], [8], (b) model-checkers [46], [33] (c) simulation tools [31], etc. Indeed, the simplicity of LUSTRE – with its underlying deterministic, clocked, structured model – makes it both attractive and versatile as a programming paradigm: It can express distributed embedded controllers, numerical computations, and complex Scade 6 [16] safety-critical systems with its support for difference equations. Moreover, the gamut of formal structures such as automata, regular expressions, temporal logic, test harnesses, synchronous observers, etc., can all be *efficiently* expressed *within* the LUSTRE model.

The only missing piece in this picture is a security model. In this paper, we seek to integrate Denning's lattice-based secure information flow (SIF) framework [18] into LUSTRE and propose a security type system. In this simple type system, (i) each stream of data is assigned a security type that is mapped to a security class from the security lattice, based on assumptions made about security types of the program variables, and (ii) the output streams from a node have security levels at least as high as the security levels of the input streams on which they depend. The rules are simple, intuitive and amenable to being incorporated into the mechanised certified compilation [36] already developed for LUSTRE [10]. The main contributions of this paper thus are: (C1) Proposal of a *security type system* which ensures secure information flow in LUSTRE. The security types are as simple as possible, which makes possible any further refinements. The main technical achievement lies in formulating appropriate rules for LUSTRE equations, with node (function) call posing particular challenges. The security types and *constraints* are stated in a *symbolic* style, thus not hard-wiring the security lattice into the rules. Based on the security typing rules, we propose a *definition of security* for LUSTRE programs. (C2) The main technical result of this paper is showing that our security type system is *sound* with respect to the *co-inductive* Stream semantics for LUSTRE, by

DST/INT/JST/P-30/2016(G) *Security in the IoT Space*, DST, Govt of India

978-1-7281-9148-5/20/$31.00 © 2020 IEEE

establishing *non-interference* [22] for well-typed programs. Security type systems have so far been proposed for imperative [18], [19] and functional-imperative languages [53]. We believe that ours is the first presentation of such a type system, and more significantly of its *soundness* with respect to the operational semantics, for a (synchronous, reactive) *data-flow language*. While our result broadly follows Volpano *et al.*'s approach [53], we believe that the adaptation to a data-flow setting is both novel and inventive. In particular, instead of a notion of *confinement checking* used to specify security in imperative paradigms, we generate and solve *constraints* for equations and programs. Thus we go beyond just *checking* that a program is secure to *inferring* constraints that suffice to ensure security.

Structure of the paper: In §II, we review the language LUSTRE and its semantics. Our presentation of the Stream semantics of LUSTRE is consistent with the CompCert encoding on a github repository [10] mentioned in [8]. §III recapitulates Denning's lattice-based model for secure information flow (SIF), and then presents the basic type system in a syntax-oriented manner. Based on these rules, we propose a definition of security for LUSTRE programs. The main results follow in §IV, where we show, using the *simple security* lemma for expressions and the *constraints* for equations and programs, that securely-typed LUSTRE programs exhibit *non-interference* with respect to the Stream semantics. The paper concludes with a discussion of the related work (§V) and directions for future work in §VI. Auxiliary definitions such as those used in the Stream semantics are presented in Appendix A.

II. A BRIEF OVERVIEW OF LUSTRE

LUSTRE [24], [44], [8] is a synchronous data-flow language used for modelling, simulating, and verifying a wide range of reactive programs including embedded controllers, safety-critical systems, communication protocols, railway signal networks, etc. In LUSTRE, a (reactive) system is represented as data streams flowing between operators and nodes in a data-flow network, *i.e.*, a synchronous analogue to Kahn Process networks [32]. The main characteristics of the language are:

1) *Declarative Style*: Programs consist of a set of *nodes*, each of which contains *defining equations*. The order in which equations are written has no effect on the semantics of the program. Equations exhibit *referential transparency*, referred to as the **Substitution Principle**.
2) *Synchronous Semantics:* Each variable and expression defines a data stream, indexed with respect to a *clock*. A clock is a stream of boolean values. A flow takes its n^{th} value on the n^{th} clock tick, *i.e.*, when the clock has value true.
3) *Deterministic Behaviour:* Program behaviours in LUSTRE are completely determined by sequences of clocked occurrences of events.
4) *Temporal Operators:* `when`, `merge` and `fby` are used to express complex clock-changing and clock-dependent behaviours.

LUSTRE has seen a steady development of its suite of tools over three decades, commencing with its introduction [11] through to formally certified compiler developments [7], [8]. In this paper, we will use a normalised form of the core syntax of LUSTRE (Figure 1), taken from [8], [1], but without the `reset` operator introduced there. (Note: LUSTRE keywords are written in `teletype` face, meta-variables in *italic* face, while semantic values and operations are in sans serif face.)

Expressions comprise variables, constants, and those constructed using unary and binary operators. In addition, an expression can be *sampled when* a *variable* takes the value k (restricted to boolean values in [8]). Control expressions are a special class of expressions that comprise the previously mentioned expressions as well as the *conditional* composition of two control expressions, and the *merge* of two control expressions based on a boolean variable. *Clocks* are special boolean valued expressions, that are either `base` or a derived (slower) clock that samples when a (boolean) *variable* takes the value k (again, restricted to booleans in [8]). We write $\vec{e}$ to denote a vector of expressions. *Equations* are of three kinds: (i) a simple definition of a program variable equated to a control expression; (ii) a program variable equated to an initialised delay-expression; and a tuple of program variables defined by a *node call*. LUSTRE being a clock-synchronous language, equations are governed by a clock parameter.

A *node* consists of a named function which takes an n-tuple of (clocked) streams as input and outputs an m-tuple of streams which are defined using a set of equations. The equations in a node may also define and use *local* variables. All variables used in a node are either explicitly classified as input, output or local variables. Each output or local program variable has a *unique* defining equation within a node. Defining equations may be recursive, provided they are well-clocked [44], [24]. In normalised LUSTRE (*i.e.*, prior to the "scheduling transform", which is beyond the scope of this paper), the order of equations is not relevant. The notions of *free* and *defined* variables in equations are as expected (definitions elided).

The text of a program in normalised LUSTRE consists of a set of *node definitions*, each of which uniquely determine a named function. There is a distinguished ("`main`") node containing the top-level equations, on which no node is dependent. Note that LUSTRE does not permit recursive node calls or cyclic dependencies. Thus the text of a well-formed LUSTRE program can be thought of as forming a directed acyclic graph G, with `main` being the unique apex node. Therefore, we can perform a topological sort of the DAG G, and give nodes dependency levels (0 is the level for nodes with no dependency on any other node).

LUSTRE maintains the following core principles:

Definition II.1 (Definition Principle)**.** *The context in which an expression e is used can have no influence on the behaviour of that expression. In particular, no information may be inferred about the input. The principle extends naturally to tuples of expressions and sets of simultaneous equations.*

Definition II.2 (Substitution Principle)**.** *An equation $x = e$*

$$
\begin{array}{lr}
e := & \textbf{(expression)} \\
\quad \mid x & \text{(variable)} \\
\quad \mid c & \text{(constant)} \\
\quad \mid \diamond\ e & \text{(unary op)} \\
\quad \mid e \oplus e & \text{(binary op)} \\
\quad \mid e\ \texttt{when}\ x = k & \text{(}k\text{-sample)} \\
d := & \textbf{(node declaration)} \\
\quad \mid \texttt{node}\ f(\vec{x}^{ck})\ \texttt{returns}\ \vec{y}^{ck} & \\
\quad\ \ \texttt{var}\ \vec{z}\ \texttt{let}\ \overrightarrow{eqn}\ \texttt{tel} &
\end{array}
\qquad
\begin{array}{lr}
ce := & \textbf{(control expression)} \\
\quad \mid e & \text{(expression)} \\
\quad \mid \texttt{merge}\ x\ ce\ ce & \text{(merge)} \\
\quad \mid \texttt{if}\ e\ \texttt{then}\ ce\ \texttt{else}\ ce & \text{(if-then-else)} \\
ck := & \textbf{(clock)} \\
\quad \mid \texttt{base} & \text{(base clk)} \\
\quad \mid ck\ \texttt{on}\ x = k & \text{(k-clock)} \\
eqn := & \textbf{(equation)} \\
\quad \mid x =_{ck} ce & \text{(defn)} \\
\quad \mid x =_{ck} c\ \texttt{fby}\ e & \text{(delay)} \\
\quad \mid \vec{x} =_{ck} f(\vec{e}) & \text{(node call)}
\end{array}
$$

Fig. 1. Normalised core LUSTRE syntax

```
--simple counter node with a reset
node Ctr(init^α1, incr^α2: int, rst^α3: bool)
    returns (n^β: int);
var fst^δ1: bool, pre_n^δ2: int;
let
    n =_base^γ if (fst^δ1 or rst^α3) then init^α1
        else pre_n^δ2 + incr^α2;
    fst   =_base^γ true^⊥ fby false^⊥;
    pre_n =_base^γ 0^⊥ fby n^β;
tel
```

specifies the program variable x and the expression e to be completely synonymous. In any context, x may be replaced by e, and vice versa. The principle extends naturally to sets of simultaneous equations.

LUSTRE has a carefully designed system of static analyses including type checking, clock checking [44] and cyclic dependency checks [24], the details of which are beyond the scope of this paper, but on which we rely for the correctness of our results. Indeed, we will consider only well-typed, well-clocked and well-formed normalised core LUSTRE programs in this paper.

A. LUSTRE *Example*

We present a small example of a LUSTRE program. (For the moment, let us ignore the superscript annotations in *blue*.)

Example II.1 (Counter). *`Ctr` is a node which takes two integer stream parameters `init` and `incr` and a boolean parameter `rst`, representing (respectively) an initial value, the increment and a reset signal stream. The output is the integer stream `n`. Two local variables are declared: the boolean stream `fst`, which is true initially and false thereafter, and the integer stream `pre_n`, which latches onto the previous value of `n`. The equation for `n` sets it to the value of `init` if either `fst` or `rst` is true, otherwise adding `incr` to `n`'s previous value (`pre_n`). `pre_n` is initialised to 0, and thereafter (using `fby`) trails the value of `n` by a clock instant. All equations here are on the same implicit "base" clock, and the calculations may be considered as occurring synchronously.*

An example run of `Ctr` is the following:

Flow	*Values*						
init	‹1›	‹2›	‹1›	‹1›	‹0›	‹2›	‹4›
incr	‹1›	‹2›	‹2›	‹3›	‹3›	‹1›	‹2›
rst	‹F›	‹F›	‹F›	‹F›	‹T›	‹F›	‹T›
fst	‹T›	‹F›	‹F›	‹F›	‹F›	‹F›	‹F›
n	‹1›	‹3›	‹5›	‹8›	‹0›	‹1›	‹4›
pre_n	‹0›	‹1›	‹3›	‹5›	‹8›	‹0›	‹1›

Example II.2 (Speedometer). *We next define another node, `SpdMtr`, using two instances of node `Ctr`. `spd` is calculated by invoking `Ctr` with suitable initial value 0 and increment `acc`, while `pos` is calculated with initial value 3 and increment `spd`. Again, both equations are on the same base clock, and the calculations are synchronous. In Example II.2, the two instances of `Ctr` are never reset.*

```
--reusing counter node
node SpdMtr(acc^α4: int)
    returns (spd^β1,pos^β2: int);
let
    spd =_base^γ2 Ctr(0^⊥,acc^α4, false^⊥);
    pos =_base^γ2 Ctr(3^⊥,spd^β1, false^⊥);
tel
```

B. *Motivation: Information flow leaks*

Currently LUSTRE does not have a security framework. Suppose we decorated variables in a LUSTRE program with security levels drawn from a Denning-style lattice. We give two simple instances of insecure expressions which can leak information implicitly. The conditional expression $\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ which, depending on whether e_0 is true or false, evaluates expression e_1 or else e_2 (all expressions are on the same clock), can leak the value of the e_0. Consider the following example, where by observing the public flow named `c`, we can learn the secret variable `b`:

```
-- b secret, c public
    c = if b then 1 else 0
```

Similarly, the expression `merge` $x\ e_1\ e_2$ – which merges, based on the value of x at each instant, the corresponding value from streams e_1 or e_2 into a single stream – also can leak the variable x's values. This is evident in the following:

```
-- x secret, c0 public
    c0 = merge x 1 0
```

Our type system aims at preventing such *implicit flows*. Further, it should be able to correctly combine the security levels of the arguments for all operators, and allow only legal flows in equational definitions of variables, node definitions and node calls.

C. Stream semantics

We now detail the semantics of LUSTRE programs in terms of co-inductively defined *streams of values*. The formulation is essentially an abstract presentation of the Velus compiler formalisation on a github repository [10] reported in [8], though we are unaware of any earlier presentation in this form. The semantic rules for the evaluation of expressions are given in Figure 2, those for clocks and clock-annotated expressions in Figure 3, and those for control expressions in Figure 4. The Velus formalisation introduces clock-annotated expressions of the form $e :: ck$ and $ce :: ck$ in order to capture the correct clock-synchronous behaviour of equations. Semantic operations are written in blue sans serif typeface. The semantic predicates assume a (co-inductive) stream *history* ($H_* : \mathit{Ident} \to \mathit{value\ Stream}$) and a clock bs and relate an expression, control expression, annotated expression or clock expression to a Stream of values. A clock is a stream of booleans (CompCert/Coq's true and false in Velus). The evaluation rules for equations, including function calls, and the semantics of node definitions are given in Figure 5. Here the semantic predicate for equations in a program (graph) G establishes *consistency* between the assumed Stream history for the program variables and the history induced by the equations, under the requisite conditions. The semantics for a node call establishes the relationship between the input and output streams.

The predicate $H_*, bs \vdash e \Downarrow_{\mathsf{e}} vs$ in Figure 2 defines the meaning of a LUSTRE expression e, with respect to a given history H_* associated with each variable and a clock bs, to be the resultant stream vs. The streams beat to the pulse of the clock, *i.e.*, some value $\langle v \rangle$ is present at instants when the clock is true, and absent (written $\langle\rangle$) when it is false. Rule (SScnst) maps a constant c via the auxiliary operation const to a stream which has value $\langle c \rangle$ present exactly when the given clock bs is true, and $\langle\rangle$ at the instants when clock bs is false. In rule (SSvar), a variable is mapped to its associated stream according to the history H_*. Unary and binary operators are applied point-wise on the argument streams, as stated in rules (SSunop) and (SSbinop). In the latter, both argument streams are assumed to pulse to the (same) given clock bs. The rule (SSwh) for e `when` $x = k$ allows projecting a stream to exactly those instants when the clock is true and the (boolean) variable x has a given value k. All auxiliary operators, defined co-inductively in Appendix A, ensure that the resulting streams have values present only when the given clock bs is true, or, in the case of when, according to a derived clock.

Likewise, the predicate $H_*, bs \vdash ck \Downarrow_{\mathsf{ck}} bs'$ in Figure 3 defines the meaning of a LUSTRE clock expression ck with respect to a given history H_* and a clock bs to be the resultant clock bs'. Coarser clocks can be defined over a given clock using the `on` construct, whenever a LUSTRE variable has the desired value and the given clock is true. The three cases (with the LUSTRE variable x having the desired value k and clock being true; with the clock being false; and with the program variable x having the complementary value and the clock being true) are listed in the rules (SSonT), (SSonA1) and (SSonA2) respectively. These rules use the auxiliary operations tl and htl, which give the tail of a stream, and the tails of streams for each variable according to a given history H_*. In rules (SSe2aeA)-(SSce2ae) for clock-annotated expressions, the output stream carries a value exactly when the clock is true.

The normalised LUSTRE language treats as special control expressions for merging streams and conditional combination. We use the co-inductively defined auxiliary stream operations merge and ite to synchronise the combination of streams according to the given clock when defining the predicate $H_*, bs \vdash ce \Downarrow_{\mathsf{ce}} vs$ in Figure 4. The control expression `merge` $x\ e_t\ e_f$ assumes that e_t's value is present and e_f's value is absent at those instances where x is true, and complementarily, when x is false, e_f's value is present and e_t's value is absent, with both being absent when x's value is absent. These conditions are enforced by the auxiliary operation merge. In contrast, the conditional `if` e `then` ce_t `else` ce_f requires all three argument streams to pulse to the same clock, as ensured by ite.

Note that the rules for all kinds of expressions employ similar predicates; when we wish to abstract from the particular kind of expression, we use a generalised predicate $H_*, bs \vdash ge \Downarrow_? vs$ that can be appropriately instanced.

The semantics of equations establish the conditions under which the assumed stream history of the defined variables is consistent with the generated semantics of the right-hand side expressions. The rule for `fby` uses the auxiliary stream operation fby that prepends a constant to a resulting stream for subexpression e, again in accordance with the tempo of a given clock.

In the case of node (function) calls, a system of equations is simultaneously checked. The rule for a node in a program graph G checks various conditions with respect to the (common) tempo of all base clocks of all the input streams, establishing the consistency of all the equations defined within the node. We use the auxiliary predicates base-of to determine the clock of the input streams, and respects-clock to check that a history pulses in accordance with that clock. All auxiliary predicates are defined in Appendix A.

We will use the following lemma about the stream semantics in section IV.

$$\frac{\mathsf{const}\ bs\ c = cs}{H_*, bs \vdash c \Downarrow_{\mathsf{e}} cs}\ (\text{SScnst}) \qquad \frac{H_*(x) = xs}{H_*, bs \vdash x \Downarrow_{\mathsf{e}} xs}\ (\text{SSvar}) \qquad \frac{H_*, bs \vdash e \Downarrow_{\mathsf{e}} es \quad \widehat{\diamond}\ es = os}{H_*, bs \vdash \diamond\ e \Downarrow_{\mathsf{e}} os}\ (\text{SSunop})$$

$$\frac{H_*, bs \vdash e_1 \Downarrow_{\mathsf{e}} es_1 \quad H_*, bs \vdash e_2 \Downarrow_{\mathsf{e}} es_2 \quad es_1 \mathbin{\widehat{\oplus}} es_2 = os}{H_*, bs \vdash e_1 \oplus e_2 \Downarrow_{\mathsf{e}} os}\ (\text{SSbinop}) \qquad \frac{H_*, bs \vdash e \Downarrow_{\mathsf{e}} es \quad H_*(x) = xs \quad \mathsf{when}\ k\ es\ xs = os}{H_*, bs \vdash e\ \mathtt{when}\ x = k \Downarrow_{\mathsf{e}} os}\ (\text{SSwh})$$

Fig. 2. Stream semantics of expressions

$$\frac{}{H_*, bs \vdash \mathtt{base} \Downarrow_{\mathsf{ck}} bs}\ (\text{SSbase}) \qquad \frac{H_*, bs \vdash ck \Downarrow_{\mathsf{ck}} (\mathsf{true} \cdot bk) \quad H_*(x) = (\langle k \rangle \cdot xs) \quad (\mathsf{htl}\ H_*), (\mathsf{tl}\ bs) \vdash ck\ \mathtt{on}\ x = k \Downarrow_{\mathsf{ck}} bs'}{H_*, bs \vdash ck\ \mathtt{on}\ x = k \Downarrow_{\mathsf{ck}} (\mathsf{true} \cdot bs')}\ (\text{SSonT})$$

$$\frac{H_*, bs \vdash ck \Downarrow_{\mathsf{ck}} (\mathsf{false} \cdot bk) \quad H_*(x) = (\langle\rangle \cdot xs) \quad (\mathsf{htl}\ H_*), (\mathsf{tl}\ bs) \vdash ck\ \mathtt{on}\ x = k \Downarrow_{\mathsf{ck}} bs'}{H_*, bs \vdash ck\ \mathtt{on}\ x = k \Downarrow_{\mathsf{ck}} (\mathsf{false} \cdot bs')}\ (\text{SSonA1}) \qquad \frac{H_*, bs \vdash e \Downarrow_{\mathsf{e}} \langle\rangle \cdot es \quad H_*, bs \vdash ck \Downarrow_{\mathsf{ck}} \mathsf{false} \cdot cs}{H_*, bs \vdash e :: ck \Downarrow_{\mathsf{e}} \langle\rangle \cdot es}\ (\text{SSe2aeA})$$

$$\frac{H_*, bs \vdash ck \Downarrow_{\mathsf{ck}} (\mathsf{true} \cdot bk) \quad H_*(x) = (\langle k \rangle \cdot xs) \quad (\mathsf{htl}\ H_*), (\mathsf{tl}\ bs) \vdash ck\ \mathtt{on}\ x = \neg k \Downarrow_{\mathsf{ck}} bs'}{H_*, bs \vdash ck\ \mathtt{on}\ x = \neg k \Downarrow_{\mathsf{ck}} (\mathsf{false} \cdot bs')}\ (\text{SSonA2}) \qquad \frac{H_*, bs \vdash e \Downarrow_{\mathsf{e}} \langle v \rangle \cdot es \quad H_*, bs \vdash ck \Downarrow_{\mathsf{ck}} \mathsf{true} \cdot cs}{H_*, bs \vdash e :: ck \Downarrow_{\mathsf{e}} \langle v \rangle \cdot es}\ (\text{SSe2ae})$$

$$\frac{H_*, bs \vdash ce \Downarrow_{\mathsf{ce}} \langle\rangle \cdot es \quad H_*, bs \vdash ck \Downarrow_{\mathsf{ck}} \mathsf{false} \cdot cs}{H_*, bs \vdash ce :: ck \Downarrow_{\mathsf{ce}} \langle\rangle \cdot es}\ (\text{SSce2aeA}) \qquad \frac{H_*, bs \vdash ce \Downarrow_{\mathsf{ce}} \langle v \rangle \cdot es \quad H_*, bs \vdash ck \Downarrow_{\mathsf{ck}} \mathsf{true} \cdot cs}{H_*, bs \vdash ce :: ck \Downarrow_{\mathsf{ce}} \langle v \rangle \cdot es}\ (\text{SSce2ae})$$

Fig. 3. Stream semantics of clocks and annotated expressions

$$\frac{H_*, bs \vdash e \Downarrow_{\mathsf{e}} es}{H_*, bs \vdash e \Downarrow_{\mathsf{ce}} es}\ (\text{SSe2ce}) \qquad \frac{H_*(x) = xs \quad H_*, bs \vdash e_t \Downarrow_{\mathsf{ce}} ts \quad H_*, bs \vdash e_f \Downarrow_{\mathsf{ce}} fs \quad \mathsf{merge}\ xs\ ts\ fs = os}{H_*, bs \vdash \mathtt{merge}\ x\ e_t\ e_f \Downarrow_{\mathsf{ce}} os}\ (\text{SSmrg})$$

$$\frac{H_*, bs \vdash e \Downarrow_{\mathsf{e}} es \quad H_*, bs \vdash ce_t \Downarrow_{\mathsf{ce}} ts \quad H_*, bs \vdash ce_f \Downarrow_{\mathsf{ce}} fs \quad \mathsf{ite}\ es\ ts\ fs = os}{H_*, bs \vdash \mathtt{if}\ e\ \mathtt{then}\ ce_t\ \mathtt{else}\ ce_f \Downarrow_{\mathsf{ce}} os}\ (\text{SSite})$$

Fig. 4. Stream semantics of control expressions

Lemma 1 (Relevant variables for expression evaluation)**.** *If* $fv(ge) \subseteq X$ *and for all* $x \in X : H_*(x) = H'_*(x)$*, then* $H_*, bs \vdash ge \Downarrow_? vs$ *iff* $H'_*, bs \vdash ge \Downarrow_? vs$*.*

III. Typing for Secure Information Flow

Denning proposed complete lattices as the appropriate mathematical model for reasoning about secure information flow [18], [19]. An information flow model $\langle N, SC, \sqsubseteq, \sqcup, \bot \rangle$ consists of a set N of all data variables/objects in the system, which are assigned security classes (typically t, possibly with subscripts) from SC, which is a finite complete lattice, partially ordered by the relation $\sqsubseteq$, and with $\sqcup$ being the *least upper-bound* (LUB) operator and $\bot$ the least element of the lattice. The intuitive reading of $t_1 \sqsubseteq t_2$ is that the security class t_1 is less secure (*i.e.*, less confidential, or dually, more trusted) than t_2, and so a flow from t_1 to t_2 is permitted.

A. Information Flow Types

We define a simple security type system, where under security type assumptions for program variables, Lustre expressions are given a *security type*, and Lustre equations induce a set of *constraints over security types*.

The set of security types ST comprises (i) security type variables, typically written as $\delta \in STV$ with subscripts or diacritical marks, standing for an arbitrary security class, and (ii) the *join* of two security types:

$$\alpha, \beta, \gamma, \mu, \nu \in ST \ ::= \ \delta\ (\in STV) \mid \alpha_1 \sqcup \alpha_2.$$

The operator $\sqcup$, interpreted as the LUB in the lattice, is intended to be associative, commutative and idempotent with $\bot$ as its identity. (Security types are written in *blue Greek*.)

ρ ranges over *constraints on security types*, which are (conjunctions of) relations of the form $\alpha \sqsubseteq \beta$. We write $\beta[\alpha/\delta]$ to denote the substitution of security type α for security

$$\frac{bs = \textsf{base-of}\ \vec{vs}\quad G(f) = g\quad H_*(g.\textbf{in}) = \vec{vs}\quad H_*(g.\textbf{out}) = \vec{ys}\quad \textsf{respects-clock}\ H_*\ bs\ g.\textbf{in}\quad \forall eq \in g.\textbf{eqs}:\ G, H_*, bs \vdash eq}{G \vdash \widehat{f}(\vec{vs}) \Downarrow \vec{ys}} \qquad \frac{H_*, bs \vdash e :: ck \Downarrow_{\textsf{ce}} H_*(x)}{G, H_*, bs \vdash x =_{ck} e}$$

$$\frac{H_*, bs \vdash e :: ck \Downarrow_{\textsf{e}} vs\quad \textsf{fby}\ c\ vs = H_*(x)}{G, H_*, bs \vdash x =_{ck} c\ \texttt{fby}\ e} \qquad \frac{H_*, bs \vdash ck \Downarrow_{\textsf{ck}} \textsf{base-of}\ \vec{vs}\quad H_*, bs \vdash \vec{e} \Downarrow_{\textsf{e}} \vec{vs}\quad G \vdash \widehat{f}(\vec{vs}) \Downarrow H_*(\vec{x})}{G, H_*, bs \vdash \vec{x} =_{ck} f(\vec{e})}$$

Fig. 5. Stream semantics of nodes and equations

type variable δ in security type β. The notation extends to substitutions on constraints, i.e., $\rho[\alpha/\delta]$.

The main reason for introducing type variables and syntax for security types rather than directly mentioning the security classes of a lattice *SC* is in order to support stating, simplifying and solving constraints *symbolically*. Not only do the rules, *e.g.*, those in section III-C and subsection III-D, become easier to state using the notion of substitution, the analysis can be framed independently of the security lattice, thereby providing a degree of abstraction. In fact, the formulation permits the *inference* of minimal constraints that suffice to ensure SIF.

We assume a typing environment $\Gamma : \mathit{Ident} \to \mathit{ST}$, a partial function that associates a security type to each free variable x in a LUSTRE program.

Let $\Delta_\Gamma = \{\delta \in \mathit{STV} \mid \delta \text{ appears in } \Gamma(x) \text{ for some } x \in \mathit{dom}(\Gamma)\}$, *i.e.*, the security type variables that appear in some security type in the *range* of Γ. Let $s : \Delta_\Gamma \to \mathit{SC}$ be a *ground instantiation* of security type variables with respect to some information flow lattice $\mathit{FM} = \langle N, \mathit{SC}, \sqsubseteq, \sqcup, \bot\rangle$. Lift s to security types and constraints in the obvious way:

$$s(\alpha_1 \sqcup \alpha_2) = s(\alpha_1) \sqcup s(\alpha_2) \qquad s(\alpha \sqsubseteq \beta) = s(\alpha) \sqsubseteq s(\beta)$$

The (lifted) composition $\Gamma \circ s : \mathit{Ident} \to \mathit{SC}$ maps program variables to security classes in the lattice.

A ground instantiation $s : \mathit{STV} \to \mathit{SC}$ *satisfies* a constraint $\alpha \sqsubseteq \beta$ in *FM* if $\mathit{FM} \models s(\alpha) \sqsubseteq s(\beta)$. A set of constraints ρ is satisfied by ground instantiation s if s satisfies each constituent constraint in ρ. A constraint $\alpha \sqsubseteq \beta$ is *satisfiable* with respect to *FM* if for *some* ground instantiation s: $\mathit{FM} \models s(\alpha) \sqsubseteq s(\beta)$. A set of constraints ρ is satisfiable in *FM* if some ground instantiation s satisfies each constituent constraint in ρ. Note that if $h : \mathit{SC} \to \mathit{SC'}$ is a structure-preserving lattice morphism, *i.e.*, $t_1 \sqsubseteq t_2 \in \mathit{SC}$ implies $h(t_1) \sqsubseteq' h(t_2) \in \mathit{SC'}$, then if ρ is satisfied by s, it is also satisfied by $s \circ h : \mathit{STV} \to \mathit{SC'}$.

B. Security Typing Rules for expressions

Expressions are type-checked with the predicates:

$$\Gamma \overset{e}{\vdash} e : \alpha, \qquad \Gamma \overset{ck}{\vdash} ck : \alpha, \quad \text{and} \quad \Gamma \overset{ce}{\vdash} ce : \alpha$$

which are read as "under the context Γ mapping variables to security types, the expression/clock/control expression $e/ck/ce$ has the security type α".

Figure 6 details rules for simple expressions and control expressions. No matter what context Γ, constants have the security type $\bot$. For variables, we look up the environment Γ. Binary ($\oplus$,`when`) and ternary (`if-then-else`, `merge`) operations on flows generate a flow with a type that is the join (LUB) of the types of the operand flows. There is an implicit dependency on the (security level of the) common clock of the operand flows for these operators. This dependence on the security level of the clock is made explicit in the judgements for equations. Figure 7 shows inference rules for clocks and clock-annotated expressions. We assume Γ maps the base clock `base` to some security variable (γ by convention). Observe that in general, the security type for any constructed expression is the join of those of its components (and the clock). The structure of all three predicates being similar, for convenience we use a generalized predicate $\Gamma \overset{?}{\vdash} ge : \alpha$ to represent a parametric analysis over the appropriate syntactic structure ge (this notation is used in stating results of §IV).

C. Security Typing Rules for Equations

The security typing rules for equations use the predicate:

$$\Gamma \overset{eqn}{\vdash} eqn\ :> \rho$$

which represents that under the security assumption context Γ, equation eqn when type-elaborated generates constraints ρ. The constraints for equations are of the form $\alpha \sqsubseteq \beta$, where β is the security type of the defined variable, and α the security type obtained from that of the defining expression joined with the clock's security type. Recall that every flow in LUSTRE is defined *exactly once* and no further constraints are placed on flows according to Definition II.1.

The rules for type-elaborating equational statements are given in Figure 8. For simple equational definitions, a constraint relating the security type of the defined variable with that of its defining (control) expression is added to the set of type constraints: The security type associated with the defined variable x is at least that of the right-hand-side, *i.e.*, β, explicitly joined with the clock's security type γ. The rule for `fby` is very similar. When considering a set of equations, we take the union of the constraints generated for each equation. Note that in LUSTRE, the order of equations is not important.

Example III.1 (Constraints from equations)**.** *With respect to Example II.1, the constraints generated for the definitions of variables* `n`, `fst` *and* `pre_n` *are* $\rho_1 := \{\gamma \sqcup \delta_1 \sqcup \alpha_3 \sqcup \alpha_1 \sqcup \delta_2 \sqcup \alpha_2 \sqsubseteq \beta\}$, $\rho_2 := \{\gamma \sqcup \bot \sqcup \bot \sqsubseteq \delta_1\}$, *and* $\rho_3 := \{\gamma \sqcup \bot \sqcup \beta \sqsubseteq \delta_2\}$ *respectively.*

$$\frac{}{\Gamma \overset{e}{\vdash} c : \bot}\ (\text{CScnst}) \qquad \frac{\alpha = \Gamma(x)}{\Gamma \overset{e}{\vdash} x : \alpha}\ (\text{CSvar}) \qquad \frac{\Gamma \overset{e}{\vdash} e : \alpha}{\Gamma \overset{e}{\vdash} \diamond\ e : \alpha}\ (\text{CSunop}) \qquad \frac{\Gamma \overset{e}{\vdash} e_1 : \alpha_1 \quad \Gamma \overset{e}{\vdash} e_2 : \alpha_2}{\Gamma \overset{e}{\vdash} e_1 \oplus e_2 : \alpha_1 \sqcup \alpha_2}\ (\text{CSbinop})$$

$$\frac{\Gamma \overset{e}{\vdash} e : \alpha \quad \beta = \Gamma(x)}{\Gamma \overset{e}{\vdash} e \texttt{ when } x = k : \alpha \sqcup \beta}\ (\text{CSwh}) \qquad \frac{\gamma = \Gamma(x) \quad \Gamma \overset{ce}{\vdash} ce_1 : \alpha_1 \quad \Gamma \overset{ce}{\vdash} ce_2 : \alpha_2}{\Gamma \overset{ce}{\vdash} \texttt{merge } x\ ce_1\ ce_2 : \gamma \sqcup \alpha_1 \sqcup \alpha_2}\ (\text{CSmrg})$$

$$\frac{\Gamma \overset{e}{\vdash} e : \alpha}{\Gamma \overset{ce}{\vdash} e : \alpha}\ (\text{CSexp}) \qquad \frac{\Gamma \overset{e}{\vdash} e : \alpha \quad \Gamma \overset{ce}{\vdash} ce_1 : \beta_1 \quad \Gamma \overset{ce}{\vdash} ce_2 : \beta_2}{\Gamma \overset{ce}{\vdash} \texttt{if } e \texttt{ then } ce_1 \texttt{ else } ce_2 : \alpha \sqcup \beta_1 \sqcup \beta_2}\ (\text{CSite})$$

Fig. 6. Security typing rules for expressions and control expressions

$$\frac{\Gamma \overset{ce}{\vdash} ce : \beta \quad \Gamma \overset{ck}{\vdash} ck : \gamma}{\Gamma \overset{ce}{\vdash} ce :: ck : \beta \sqcup \gamma}\ (\text{CSace}) \qquad \frac{\gamma = \Gamma(\texttt{base})}{\Gamma \overset{ck}{\vdash} \texttt{base} : \gamma}\ (\text{CSbase}) \qquad \frac{\Gamma \overset{ck}{\vdash} ck : \gamma \quad \beta = \Gamma(x)}{\Gamma \overset{ck}{\vdash} ck \texttt{ on } x = k : \gamma \sqcup \beta}\ (\text{CSck})$$

Fig. 7. Security typing rules for clocks

$$\frac{\alpha = \Gamma(x) \quad \Gamma \overset{e}{\vdash} ce : \beta \quad \Gamma \overset{ck}{\vdash} ck : \gamma}{\Gamma \overset{eqn}{\vdash} x =_{ck} ce \ :> \{\beta \sqcup \gamma \sqsubseteq \alpha\}}\ (\text{CSeqn}) \qquad \frac{\alpha = \Gamma(x) \quad \Gamma \overset{e}{\vdash} e : \beta \quad \Gamma \overset{ck}{\vdash} ck : \gamma}{\Gamma \overset{eqn}{\vdash} x =_{ck} c \texttt{ fby } e \ :> \{\beta \sqcup \gamma \sqsubseteq \alpha\}}\ (\text{CSfby})$$

$$\frac{\Gamma \overset{eqn}{\vdash} eqn \ :> \rho \quad \Gamma \overset{eqn}{\vdash} eqns \ :> \rho'}{\Gamma \overset{eqn}{\vdash} eqn; eqns \ :> \rho \cup \rho'}\ (\text{CSeqns})$$

Fig. 8. Security typing rules for equations

Security Typing Rules for Function Calls: Node instantiation uses the node's security signature (explained in greater detail in subsection III-D) to define the constraints. Let us assume that the node defining f is such that it takes in n input streams and produces m output streams. A node security signature is of the form

$$\overset{Node}{\vdash} \texttt{Node f } (\vec{\alpha})^{\gamma} \xrightarrow{\rho} \vec{\beta}$$

where the α_i are type *variables* for the security types of the respective input streams, β_j are type *variables* for the security types of the output streams, γ is the security type variable for the (common) clock of the input streams, and ρ the security type constraints obtained by type-elaborating the defining equations in the node. The rule for node call is:

$$\frac{\begin{array}{c}\overset{Node}{\vdash} \texttt{Node f } (\vec{\alpha})^{\gamma} \xrightarrow{\rho} \vec{\beta} \\ \Gamma \overset{e}{\vdash} \vec{e} : \vec{\alpha'} \quad \vec{\beta'} = \Gamma(\vec{x}) \quad \Gamma \overset{ck}{\vdash} ck : \gamma' \\ \rho' = \rho[\gamma'/\gamma][\vec{\alpha'}/\vec{\alpha}][\vec{\beta'}/\vec{\beta}]\end{array}}{\Gamma \overset{eqn}{\vdash} \vec{x} =_{ck} f(e_1, \ldots e_n) \ :> \rho'}\ (\text{CScall})$$

The rule can be understood as saying the following: find the security types $(\alpha'_1, \ldots \alpha'_n)$ of all input flows $(e_1, \ldots e_n)$, and from the environment Γ, find the expected security types $\vec{\beta'}$ of the output flows, then instantiate these in a copy of the constraints from the node security signature.

The node call rule can be formulated in this simple and modular manner since LUSTRE does not allow recursive node calls and cyclic dependencies. Moreover, all variables in a node definition are explicitly accounted for as input and output parameters or local variables. Of course, the rule relies on the correct formulation of the node signature and the attendant constraints. In particular, when local variables are themselves defined using a node call, the β'_i will always be type variables according to the prevailing type environment.

In subsection III-D, we will illustrate the rule (CScall) with respect to the definitions of `spd` and `pos` in node `SpdMtr` of Example II.2.

D. Node definition

Node definitions induce *node signatures* of the form:

$$\overset{Node}{\vdash} \texttt{Node f } (\vec{\alpha})^{\gamma} \xrightarrow{\rho} \vec{\beta}$$

The node security signature is obtained as follows. From the node definition, create type assumptions by associating type variables for the input, output, clock and local identifiers. Then

$$\frac{}{\rho' = \mathsf{simplify}\ \rho\ [\,]}$$

$$\frac{\rho' = \mathsf{simplify}\ \rho[\nu/\delta]\ \vec{\delta}}{\rho' = \mathsf{simplify}\ (\rho \cup \{\nu \sqsubseteq \delta\})\ [\delta; \vec{\delta}]} \quad \delta \text{ not in } \nu$$

$$\frac{\rho' = \mathsf{simplify}\ \rho[\nu/\delta]\ \vec{\delta}}{\rho' = \mathsf{simplify}\ (\rho \cup \{\nu \sqcup \delta\ \sqsubseteq\ \delta\})\ [\delta; \vec{\delta}]} \quad \delta \text{ not in } \nu$$

Fig. 9. Eliminating local variables' security type constraints

type-elaborate the equations to obtain constraints, which are then turned into a canonical form. This is formalised as:

$$\left\{ \begin{array}{c} \mathsf{name} = \ \mathtt{f};\ \mathsf{in} = \ \overrightarrow{x :: ck};\ \mathsf{var} = \ \vec{z};\\ \mathsf{out} = \ \vec{y};\ \mathsf{eqs} = \ \overrightarrow{eqn} \end{array} \right\} \in G$$

$$\Gamma_F := \{\vec{x} \mapsto \vec{\alpha}, \vec{y} \mapsto \vec{\beta}, ck \mapsto \gamma\}$$

$$\Gamma_L := \{\vec{z} \mapsto \vec{\delta}\}$$

$$\frac{\Gamma_F \cup \Gamma_L \overset{eqn}{\vdash} \overrightarrow{eqn}\ :> \rho' \quad \rho = \mathsf{simplify}\ \rho'\ \vec{\delta}}{\overset{Node}{\vdash}\ \mathtt{Node\ f}\ (\vec{\alpha})^{\gamma} \xrightarrow{\rho} \vec{\beta}} \text{(CSndef)}$$

where $\alpha_1, \ldots, \alpha_n, \delta_1, \ldots \delta_k, \beta_1, \ldots \beta_m, \gamma$ are distinct *type variables* (for clarity, assume these are all *fresh* variables).

Local variables in LUSTRE are used as aliases for expressions on input/output variables. Observe that the δ_i are the security type variables assigned to the local variables in a node, and so should not appear in the security signature of a node. Since there will be exactly one defining equation for any local variable z_i, and since in Γ_L we introduce a *fresh* type variable δ_i for local variable z_i, note that in constraints ρ', there will be exactly one constraint in which δ_i is on the right, and this is of the form $\nu_i \sqsubseteq \delta_i$.

To eliminate the δ_i, the constraints are turned into canonical form by the auxiliary program $\mathsf{simplify}\ \rho\ \vec{\delta}$, defined in Figure 9. This serially eliminates the type variables $\vec{\delta}$ assigned to the local variables in the node definition. Observe that constraints of the form $\{\nu \sqcup \delta\ \sqsubseteq\ \delta\}$ can arise due to recursion in equations. The notation $[\delta; \vec{\delta}]$ denotes a sequence of distinct type variables consisting of first δ followed by the remaining sequence $\vec{\delta}$.

We noted above that the rules for equations introduce exactly one constraint of the form $\nu \sqsubseteq \delta$ for each such δ, since each program variable has a unique defining equation. If all such security type variables have been eliminated in the simplification process, then the constraints are in canonical form. We use an arbitrary but fixed order on type variables and the property that $\sqcup$ is idempotent, associative and commutative, and has $\bot$ as identity, to transform any security type appearing in a constraint to a suitable canonical representation. Observe that due to the unique definition property noted above, the right sides of constraints that are type variables will remain as type variables during this simplification.

For Example III.1, $\mathsf{simplify}\ (\rho_1 \cup \rho_2 \cup \rho_3)\ [\delta_1; \delta_2]$ yields $\rho = \{\gamma \sqcup \alpha_1 \sqcup \alpha_2 \sqcup \alpha_3 \sqsubseteq \beta\}$. Thus the node signature for `Ctr` is

$$\overset{Node}{\vdash}\ \mathtt{Node\ Ctr}\ (\alpha_1, \alpha_2, \alpha_3)^{\gamma} \xrightarrow{\rho} \beta$$

For Example II.2, the constraints generated for the equations defining `spd` and `pos` are $\rho_4 := \{\gamma_1 \sqcup \bot \sqcup \alpha_4 \sqcup \bot \sqsubseteq \beta_1\}$ and $\rho_5 := \{\gamma_1 \sqcup \bot \sqcup \beta_1 \sqcup \bot \sqsubseteq \beta_2\}$ respectively. $\rho_4 \cup \rho_5$ simplifies to $\{\gamma_1 \sqcup \alpha_4 \sqsubseteq \beta_1, \gamma_1 \sqcup \beta_1 \sqsubseteq \beta_2\}$. Since $\gamma_1 \sqsubseteq \beta_1$, the latter constraint is equivalent to $\beta_1 \sqsubseteq \beta_2$.

Lemma 2 (Correctness of $\mathsf{simplify}\ \rho\ \vec{\delta}$)**.** *Let ρ be a set of constraints such that for a security type variable δ, there is at most one constraint of the form $\mu \sqsubseteq \delta$. Let s be a ground instantiation of security type variables in an information flow lattice FM such that ρ is satisfied by s.*

1) *If $\rho = \rho_1 \cup \{\nu \sqsubseteq \delta\}$, where variable δ is not in ν, then $\rho_1[\nu/\delta]$ is satisfied by s. (Assume disjoint union.)*
2) *If $\rho = \rho_1 \cup \{\nu \sqcup \delta \sqsubseteq \delta\}$, where variable δ is not in ν, then $\rho_1[\nu/\delta]$ is satisfied by s. (Assume disjoint union.)*

PROOF SKETCH. Note that ρ_1 is satisfied by s, and that δ appears to the right of $\sqsubseteq$ in only one constraint. Suppose $\beta_1 \sqsubseteq \beta_2$ is a constraint in ρ_1, with variable δ appearing in β_1. Since $FM \models s(\nu) \sqsubseteq s(\delta)$, by transitivity and monotonicity of s with respect to $\sqcup$: $s(\beta_1[\nu/\delta]) \sqsubseteq s(\beta_1) \sqsubseteq s(\beta_2)$. □

Lemma 3 (Security of Node Calls)**.** *Suppose g is a node in the graph G of a* LUSTRE *program ($g = G(f)$ for some f), which has security signature*

$$\overset{Node}{\vdash}\ \mathtt{Node\ f}\ (\vec{\alpha})^{\gamma} \xrightarrow{\rho} \vec{\beta}.$$

and also suppose

$$\Gamma \overset{eqn}{\vdash} \vec{x} =_{ck} f(e_1, \ldots e_n)\ \ :> \rho_1$$

Let s be a ground instantiation of type variables such that for some security classes $\vec{t}, \vec{u'}, w \in SC$: $s(\vec{\alpha'}) = \vec{t}$ where $\Gamma \overset{e}{\vdash} \vec{e} : \vec{\alpha'}$, $s(\vec{\nu}) = \vec{u'}$ where $\Gamma(\vec{x}) = \vec{\nu}$, and $s(\gamma) = w$ where $\Gamma \overset{ck}{\vdash} ck : \gamma$.

Now, if ρ is satisfied by the ground instantiation $\{\vec{\alpha} \mapsto \vec{t}, \vec{\beta} \mapsto \vec{u}, \gamma \mapsto w\}$ where $\vec{u} \sqsubseteq \vec{u'}$ (point-wise ordering on the tuples), then ρ_1 is satisfied by s.

Definition III.1 (Node Security)**.** *Let g be a node in the graph G of a* LUSTRE *program ($g = G(f)$ for some f), which has security signature*

$$\overset{Node}{\vdash}\ \mathtt{Node\ f}\ (\vec{\alpha})^{\gamma} \xrightarrow{\rho} \vec{\beta}.$$

Let s be a ground instantiation that maps the security type variables in the set $\{(\alpha_1, \ldots \alpha_n)\} \cup \{(\beta_1, \ldots \beta_m)\} \cup \{\gamma\}$ to the security classes SC of an information flow lattice FM. Node g is secure with respect to s if

1) *ρ is satisfied by s;*
2) *For each node g' on which g is directly dependent, g' is secure with respect to the ground instantiations as given by Lemma 3 for each call to g' in g.*

IV. Soundness of the Type System

We establish the soundness of the type system by adapting the approach of Volpano *et al* [53] to a data-flow setting. The novelty of the approach is to dispense with the usual notion of *confinement checking* but instead to generate and solve security type constraints.

The Simple Security Lemma for expressions (respectively, control expressions and clock expressions says: "if, under given security assumptions for the free program variables, the type system gives a general expression ge (expression, control expression, clock expression) a security type α, then all variables which may have been read in evaluating the expression have a security level that is α or lower".

Lemma 4 (Simple Security)**.** *For any general expression ge and security type assumption Γ, if $\Gamma \overset{?}{\vdash} ge : \alpha$, then for all $x \in fv(ge) : \Gamma(x) \sqsubseteq \alpha$.*

Proof sketch. By induction on the structure of ge. Constants, variables and `base` are the base cases. The result is immediate from the fact that in the rules for $\Gamma \overset{?}{\vdash} ge : \alpha$, the security level of a (generalised) expression is the join of the security levels of the component sub-expressions. □

Definition IV.1 ($(\sqsubseteq t)$-projected Stream)**.** *Suppose $t \in SC$ is a security class in FM. Let X be a set of program variables, Γ be security type assumptions for variables in X, and s be a ground instantiation, i.e., $\Gamma \circ s$ maps variables in X to security classes in SC. Let us define $X_{\sqsubseteq t} = \{x \in X \mid (\Gamma \circ s)(x) \sqsubseteq t\}$. Let H_* be a Stream history such that $X \subseteq dom(H_*)$. Define $H_*|_{X_{\sqsubseteq t}}$ as the projection of H_* to $X_{\sqsubseteq t}$, i.e., restricted to those variables that are at security level t or lower:*

$$H_*|_{X_{\sqsubseteq t}}(x) = H_*(x) \quad \text{for } x \in X_{\sqsubseteq t}.$$

Theorem 5 (Non-interference)**.** *Let $g \in G$ be a node with security signature*

$$\overset{Node}{\vdash} \texttt{Node f } \vec{\alpha}^{\gamma} \xrightarrow{\rho} \vec{\beta}$$

which is secure with respect to ground instantiation s of the type variables.
Let eqs be the set of equations in g. Let $X = fv(eqs) - dv(eqs)$, i.e., the input variables in eqs.
Let $V = fv(eqs) \cup dv(eqs)$, i.e., the input, output and local variables.
Let Γ (and s) be such that $\Gamma \overset{eqn}{\vdash} eqs :> \rho$ and ρ is satisfied by s. Let $t \in SC$ be any security level. Let bs be a given (base) clock stream.
Let H_* and H'_* be such that

1) *for all $eq \in eqs$: $G, H_*, bs \vdash eq$ and $G, H'_*, bs \vdash eq$, i.e., both H_* and H'_* are consistent Stream histories on each of the equations.*
2) *$H_*|_{X_{\sqsubseteq t}} = H'_*|_{X_{\sqsubseteq t}}$, i.e., H_* and H'_* agree on the input variables which are at a security level t or below.*

Then $H_*|_{V_{\sqsubseteq t}} = H'_*|_{V_{\sqsubseteq t}}$, i.e, H_* and H'_* agree on all variables of the node that are given a security level t or below.

Proof. The proof is by induction on the dependency level of $g \in G$. For level 0 nodes, the only equations are of the form $x =_{ck} ce$ and $x =_{ck} c$ `fby` e. We first consider only single equations. Consider the case when $x \in X_{\sqsubseteq t}$ (the other case does not matter). From the rules (CSeqn) and (CSfby), we have $\beta \sqcup \gamma \sqsubseteq \alpha$, and consequently $s(\beta \sqcup \gamma) \sqsubseteq s(\alpha) \sqsubseteq t$. By Lemma 4, $fv(ge) \subseteq X_{\sqsubseteq t}$ (otherwise we would contradict $s(\alpha) \sqsubseteq t$). So by Lemma 1: $H_*, bs \vdash ge \Downarrow_? vs$ iff $H'_*, bs \vdash ge \Downarrow_? vs$. Therefore by the rules in Figure 5, $H_*(x) = H'_*(x)$.

Since the constraints of each equation must be satisfied by s, the result extends in a straightforward way to sets of equations. Thus we have established the result for nodes at dependency level 0.

In the general case, we assume that the result holds for all nodes up to a dependency level k, and now consider a node at level $k+1$.

There can now be 3 forms of equations: $x =_{ck} ce$ and $x =_{ck} c$ `fby` e (as before), and node calls. For any of the two simple cases of equation, the proof follows the reasoning given above for level 0 nodes.

We now consider the case of node call equations $\vec{x} =_{ck} f'(\vec{e})$. Suppose $x_i \in \{\vec{x}\}$. If $(\Gamma \circ s)(x_i) \not\sqsubseteq t$, there is nothing to show. So we only need to consider the case where $s(\nu_i) \sqsubseteq t$.

Since g is secure wrt s, by Definition III.1, each call to a node g' at a dependency level $\leq k$ is secure with respect to the ground instantiation specified in Lemma 3. Therefore, by invoking the induction hypothesis on g' and the Stream semantics rules in Figure 5, let us consider the Stream histories H_* and H'_* augmented to include the flows on variables of this instance of g'. Let us call these $H_*^{+g'}$ and $H_*'^{+g'}$. For the corresponding output variable y'_j of security type β''_j in node g' on which x_i depends, since $\beta''_j \sqsubseteq \nu_i$ we have: $H_*^{+g'}(y'_j) = H_*'^{+g'}(y'_j)$. Whence by the rules in Figure 5: $H_*(x_i) = H'_*(x_i)$. □

V. Related Work

Security type systems: Denning's seminal paper [18] proposed complete lattices as the appropriate structure for information flow analyses. The subsequent paper [19] presented static analysis frameworks for certifying secure information flow. A gamut of secure flow analyses were based on these foundations.

Only much later did Volpano *et al.* [53] provide a security type system with a semantic soundness result by showing that security-typed programs exhibit non-interference [22]. Type systems remain a powerful way of analysing program behaviour, particularly secure information flow. For instance, the Jif compiler [43] for the JFlow language [41] (based on Java) not only checks for IFC leaks but also deals with *declassification*, using the *Decentralised model* of data ownership [42]. Matos *et al.* proposed a *synchronous reactive* extension of Volpano's imperative framework. Their language is at a lower level than Lustre, and has explicit synchronization

primitives for broadcast signals, suspension, preemption and scheduling of concurrent threads. While they employ the notion of *reactive bisimulation* to deal with concurrency, the techniques employed closely follow Volpano's formulation of the type system (which use *var* and *cmd* types) and a reduction semantics (necessitating a subject reduction theorem). In contrast, we are able to leverage the declarative elegance and simplicity of LUSTRE to present a far simpler type system and its soundness proof in terms of LUSTRE's co-inductive stream semantics.

Semantics and logics: Non-interference [22] is considered a standard semantic notion for security although other notions of semantic correctness have been proposed, *e.g.*, [6]. Non-interference is a typical *hyperproperty* [15], *i.e.*, a set of sets of program traces. Clarkson *et al.* [14] have presented temporal logics *HyperLTL, HyperCTL**, for verification of hyperproperties.

Beyond type systems: Zanotti [54] proposed an abstract interpretation framework similar to the earlier work from Volpano *et al.*, but strictly more general in its applicability. Hunt and Sands [29] extended type-based IFC checking with flow-sensitivity. In PARAGON, a Java-based language proposed by Broberg *et al.* [9], one can additionally handle the runtime tracking of typestate. However, in general, type-based techniques can exhibit imprecision as they lack flow and context sensitivity and do not systematically handle unstructured control flow and exceptions in programs. Hammer and Snelting [26] proposed the usage of program dependence graphs (PDGs) to offer a flow-, context- and object-sensitive analysis to detect IFC leaks. Livshits *et al.* [39] use data propagation graphs to automatically infer explicit information flow leaks.

Runtime techniques: Dynamic analyses provide greater precision, particularly in systems which rely on dynamic typing, when static dependency graph or type-based approaches are not adequate. Shroff *et al.* [51] have proposed dynamic tracking of dependencies to analyse noninterference. Austin and Flanagan [2], [3] have proposed dynamic checks to efficiently detect implicit flows based on the *no-sensitive-upgrade* semantics and the *permissive-upgrade* semantics. Their subsequent work [4] addressed limitations in the semantics due to which executions where implicit flows cannot be tracked are prematurely terminated.

IFC analyses in hardware and systems: We refer to but do not further discuss here work addressing IFC analyses in hardware systems, *e.g.* [52], [37], [56], [21], in programming languages [49], [45], [38], [47], in operating systems, [34], [55], [13], [20], [47], and in databases [50]. In the context of embedded systems, it will be interesting to see how our higher-level LUSTRE-based approach compares with lower-level secure hardware description languages such as SecVerilog [56] and ChiselFlow [21] in which fine-grained security policies can be expressed.

VI. CONCLUSIONS

We have presented a simple security type system for a synchronous reactive data-flow language, and shown its semantic soundness with respect to the language's stream semantics in the form of non-interference. Type systems for security and their soundness proofs have traditionally been proposed for imperative programming languages and for functional languages with imperative features [5], [28], [53]. Declarative data-flow languages such as LUSTRE pose interesting issues, since there is a reactive transformation between infinite input and output streams within a clock instant, and there are interesting clock dependencies. Since the evaluation of expressions is synchronous, within a clock tick, the usual operational rules that account for control constructs are of less importance. Further, in the source program, the syntactic order of equations is irrelevant to the semantics of programs. The traditional issues of control flow and termination cannot be used in the same way, and indeed the results such as the *confinement lemma* and *subject reduction*, so central to Volpano et al's formulation [53], become less important in the streams semantic model. Non-interference requires a novel re-interpretation to handle possibly recursively defined flows, and to cater to the infinite stream semantics. On the other hand, the simple and elegant semantics of LUSTRE, particularly that all variables have unique definitions and that node calls are not recursive, greatly simplifies our formulation of the type system, the notion of security and the non-interference proof.

We are currently formalising our results in Coq and integrating them into the CompCert-Velus efforts [36], [10]. (A discussion of details is omitted from this paper for lack of space.) We are also currently extending our analysis to the translation of LUSTRE to an imperative language [7], [8]. Our preliminary results indicate that the *instantaneous semantics* for LUSTRE provides the necessary scaffolding in showing that our notions of security and non-interference are mapped by the translation to their traditional counterparts in the imperative language setting.

REFERENCES

[1] AUGER, C. *Certified compilation of SCADE / LUSTRE.* Theses, Université Paris Sud - Paris XI, Feb. 2013.

[2] AUSTIN, T. H., AND FLANAGAN, C. Efficient Purely-Dynamic Information Flow Analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security* (New York, NY, USA, 2009), PLAS '09, Association for Computing Machinery, p. 113–124.

[3] AUSTIN, T. H., AND FLANAGAN, C. Permissive Dynamic Information Flow Analysis. In *Proceedings of the 2010 Workshop on Programming Languages and Analysis for Security, PLAS 2010, Toronto, ON, Canada, 10 June, 2010* (New York, NY, USA, 2010), PLAS '10, Association for Computing Machinery.

[4] AUSTIN, T. H., SCHMITZ, T., AND FLANAGAN, C. Multiple Facets for Dynamic Information Flow with Exceptions. *ACM Trans. Program. Lang. Syst. 39*, 3 (May 2017).

[5] BARTHE, G., D'ARGENIO, P. R., AND REZK, T. Secure Information Flow by Self-Composition. In *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.* (June 2004), pp. 100–114.

[6] BOUDOL, G. Secure Information Flow as a Safety Property. In *Formal Aspects in Security and Trust, 5th International Workshop, FAST 2008, Malaga, Spain, October 9-10, 2008, Revised Selected Papers* (2008), vol. 5491 of *Lecture Notes in Computer Science*, Springer, pp. 20–34.

[7] BOURKE, T., BRUN, L., DAGAND, P.-E., LEROY, X., POUZET, M., AND RIEG, L. A Formally Verified Compiler for Lustre. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2017), PLDI 2017, Association for Computing Machinery, p. 586–601.

[8] BOURKE, T., BRUN, L., AND POUZET, M. Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset. *Proc. ACM Program. Lang. 4*, POPL (Dec. 2019).

[9] BROBERG, N., VAN DELFT, B., AND SANDS, D. Paragon for Practical Programming with Information-Flow Control. In *Programming Languages and Systems* (Cham, 2013), C.-c. Shan, Ed., Springer International Publishing, pp. 217–232.

[10] BRUN, L., BOURKE, T., AND POUZET, M. Vélus compiler repository. https://github.com/INRIA/velus, 2020. Accessed: 2020-01-20.

[11] CASPI, P., PILAUD, D., HALBWACHS, N., AND PLAICE, J. A. LUSTRE: A Declarative Language for Programming Synchronous Systems. In *Proc. 14th Symposium on Principles of Programming Languages (POPL'87). ACM* (1987).

[12] CHECKOWAY, S., MCCOY, D., KANTOR, B., ANDERSON, D., SHACHAM, H., SAVAGE, S., KOSCHER, K., CZESKIS, A., ROESNER, F., AND KOHNO, T. Comprehensive Experimental Analyses of Automotive Attack Surfaces. In *Proceedings of the 20th USENIX Conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association, pp. 6–6.

[13] CHENG, W., PORTS, D. R. K., SCHULTZ, D. A., POPIC, V., BLANKSTEIN, A., COWLING, J. A., CURTIS, D., SHRIRA, L., AND LISKOV, B. Abstractions for Usable Information Flow Control in Aeolus. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012* (2012), pp. 139–151.

[14] CLARKSON, M. R., FINKBEINER, B., KOLEINI, M., MICINSKI, K. K., RABE, M. N., AND SÁNCHEZ, C. Temporal Logics for Hyperproperties. In *Principles of Security and Trust* (Berlin, Heidelberg, 2014), M. Abadi and S. Kremer, Eds., Springer Berlin Heidelberg, pp. 265–284.

[15] CLARKSON, M. R., AND SCHNEIDER, F. B. Hyperproperties. *J. Comput. Secur. 18*, 6 (Sept. 2010), 1157–1210.

[16] COLAÇO, J., PAGANO, B., AND POUZET, M. SCADE 6: A Formal Language for Embedded Critical Software Development (invited paper). In *2017 International Symposium on Theoretical Aspects of Software Engineering (TASE)* (Sep. 2017), pp. 1–11.

[17] CRANFORD, N. IoT security concerns prompt massive pacemaker recall. https://enterpriseiotinsights.com/20170901/abbott-recalls-almost-50000-pacemakers-due-to-security-concerns-tag27. Accessed: 2020-05-02.

[18] DENNING, D. E. A Lattice Model of Secure Information Flow. *Commun. ACM 19*, 5 (May 1976), 236–243.

[19] DENNING, D. E., AND DENNING, P. J. Certification of Programs for Secure Information Flow. *Commun. ACM 20*, 7 (July 1977), 504–513.

[20] EFSTATHOPOULOS, P., KROHN, M. N., VANDEBOGART, S., FREY, C., ZIEGLER, D., KOHLER, E., MAZIÈRES, D., KAASHOEK, M. F., AND MORRIS, R. T. Labels and Event Processes in the Asbestos Operating System. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005* (2005), pp. 17–30.

[21] FERRAIUOLO, A., ZHAO, M., MYERS, A. C., AND SUH, G. E. Hyperflow: A Processor Architecture for Nonmalleable, Timing-Safe Information Flow Security. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018* (2018), pp. 1583–1600.

[22] GOGUEN, J. A., AND MESEGUER, J. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982* (1982), IEEE Computer Society, pp. 11–20.

[23] GRAHAM, J. Now a smart lightbulb system got hacked. https://www.usatoday.com/story/tech/2020/02/05/how-to-avoid-smart-lights-getting-hacked/4660430002/, 2020. Accessed: 2020-02-25.

[24] HALBWACHS, N., CASPI, P., RAYMOND, P., AND PILAUD, D. The Synchronous Data Flow Programming Language LUSTRE. *Proceedings of the IEEE 79*, 9 (Sep. 1991), 1305–1320.

[25] HALBWACHS, N., LAGNIER, F., AND RAYMOND, P. Synchronous Observers and the Verification of Reactive Systems. In *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93, Twente* (1993), Springer Verlag.

[26] HAMMER, C., AND SNELTING, G. Flow-sensitive, Context-sensitive, and Object-sensitive Information Flow Control based on program dependence graphs. *International Journal of Information Security 8*, 6 (Dec. 2009), 399–422.

[27] HARDIGREE, M. Carshark software lets you hack into, control and kill any car. https://jalopnik.com/carshark-software-lets-you-hack-into-control-and-kill-5539181, 2014. Accessed: 2020-01-20.

[28] HEINTZE, N., AND RIECKE, J. G. The SLam Calculus: Programming with Secrecy and Integrity. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1998), POPL '98, Association for Computing Machinery, p. 365–377.

[29] HUNT, S., AND SANDS, D. On Flow-Sensitive Security Types. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programm ing Languages* (New York, NY, USA, 2006), POPL '06, Association for Computing Machinery, p. 79–90.

[30] INTERNET OF THINGS GLOBAL STANDARDS INITIATIVE. Internet of things global standards initiative. https://www.itu.int/en/ITU-T/gsi/iot, 2015. Accessed: 2020-01-20.

[31] JAHIER, E. *The Lurette V2 User guide*, V2 ed. Verimag, October 2015. http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lurette/doc/lurette-man.pdf.

[32] KAHN, G. The Semantics of a Simple Language for Parallel Programming. In *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974* (1974), J. L. Rosenfeld, Ed., North-Holland, pp. 471–475.

[33] KIND 2 GROUP. *Kind 2 User Documentation*, version 1.2.0 ed. Department of Computer Science, The University of Iowa, April 2020. https://kind.cs.uiowa.edu/kind2_user_doc/doc.pdf.

[34] KROHN, M. N., YIP, A., BRODSKY, M. Z., CLIFFER, N., KAASHOEK, M. F., KOHLER, E., AND MORRIS, R. T. Information Flow Control for Standard OS Abstractions. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007* (2007), pp. 321–334.

[35] LANDIN, P. J. The next 700 Programming Languages. *Commun. ACM 9*, 3 (1966), 157–166.

[36] LEROY, X. Formal verification of a realistic compiler. *Commun. ACM 52*, 7 (2009), 107–115.

[37] LI, X., KASHYAP, V., OBERG, J. K., TIWARI, M., RAJARATHINAM, V. R., KASTNER, R., SHERWOOD, T., HARDEKOPF, B., AND CHONG, F. T. Sapper: A Language for Hardware-Level Security Policy Enforcement. *SIGARCH Comput. Archit. News 42*, 1 (Feb. 2014), 97–112.

[38] LIU, J., ARDEN, O., GEORGE, M. D., AND MYERS, A. C. Fabric: Building open distributed systems securely by construction. *Journal of Computer Security 25*, 4-5 (2017), 367–426.

[39] LIVSHITS, B., NORI, A. V., RAJAMANI, S. K., AND BANERJEE, A. Merlin: Specification Inference for Explicit Information Flow Problems. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Im plementation* (New York, NY, USA, 2009), PLDI '09, Association for Computing Machinery, p. 75–86.

[40] MARIN, E., SINGELÉE, D., GARCIA, F. D., CHOTHIA, T., WILLEMS, R., AND PRENEEL, B. On the (in)Security of the Latest Generation Implantable Cardiac Defibrillators and How to Secure Them. In *Proceedings of the 32nd Annual Conference on Computer Security Applications* (New York, NY, USA, 2016), ACSAC '16, ACM, pp. 226–236.

[41] MYERS, A. C. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA, Jan. 1999), POPL '99, Association for Computing Machinery, pp. 228–241.

[42] MYERS, A. C., AND LISKOV, B. A Decentralized Model for Information Flow Control. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1997), SOSP '97, Association for Computing Machinery, p. 129–142.

[43] MYERS, A. C., ZHENG, L., ZDANCEWIC, S., CHONG, S., AND NYSTROM, N. Jif 3.0: Java information flow. http://www.cs.cornell.edu/jif.

[44] PLAICE, J. The LUSTRE Synchronous Dataflow Programming Language: Design and Semantics. *Annals of the New York Academy of Sciences 661* (12 2006), 118 – 151.

[45] POTTIER, F., AND SIMONET, V. Information Flow Inference for ML. *ACM Trans. Program. Lang. Syst. 25*, 1 (Jan. 2003), 117–158.

[46] RAYMOND, P. *Synchronous Program Verification with Lustre/Lesar*. Wiley, 2010, pp. 171 – 206.

[47] ROY, I., PORTER, D. E., BOND, M. D., MCKINLEY, K. S., AND WITCHEL, E. Laminar: Practical Fine-Grained Decentralized Information Flow Control. In *Proceedings of the 2009 ACM SIGPLAN*

Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009 (2009), pp. 63–74.

[48] RUSHBY, J. M. The Versatile Synchronous Observer. In *Formal Methods: Foundations and Applications - 15th Brazilian Symposium, SBMF 2012, Natal, Brazil, September 23-28, 2012. Proceedings* (2012), vol. 7498 of *Lecture Notes in Computer Science*, Springer, p. 1.

[49] SABELFELD, A., AND MYERS, A. C. Language-based Information-Flow Security. *IEEE Journal on Selected Areas in Communications 21*, 1 (2003), 5–19.

[50] SCHULTZ, D. A., AND LISKOV, B. IFDB: Decentralized Information Flow Control for Databases. In *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013* (2013), pp. 43–56.

[51] SHROFF, P., SMITH, S., AND THOBER, M. Dynamic Dependency Monitoring to Secure Information Flow. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium* (USA, 2007), CSF '07, IEEE Computer Society, p. 203–217.

[52] TIWARI, M., OBERG, J. K., LI, X., VALAMEHR, J., LEVIN, T., HARDEKOPF, B., KASTNER, R., CHONG, F. T., AND SHERWOOD, T. Crafting a Usable Microkernel, Processor, and I/O System with Strict and Provable Information Flow Security. In *Proceedings of the 38th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2011), ISCA '11, Association for Computing Machinery, p. 189–200.

[53] VOLPANO, D., IRVINE, C., AND SMITH, G. A Sound Type System for Secure Flow Analysis. *J. Comput. Secur. 4*, 2–3 (Jan. 1996), 167–187.

[54] ZANOTTI, M. Security Typings by Abstract Interpretation. In *Proceedings of the 9th International Symposium on Static Analysis* (Berlin, Heidelberg, 2002), SAS '02, Springer-Verlag, p. 360–375.

[55] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making Information Flow Explicit in HiStar. In *7th Symposium on Operating Systems Design and Implementation (OSDI'06), November 6-8, Seattle, WA, USA* (2006), pp. 263–278.

[56] ZHANG, D., WANG, Y., SUH, G. E., AND MYERS, A. C. A Hardware Design Language for Timing-Sensitive Information-Flow security. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey, Mar. 2015), ASPLOS '15, Association for Computing Machinery, pp. 503–516.

APPENDIX

The definitions of the auxiliary semantic stream predicates when, const, merge, ite are given in Figure 10.

All auxiliary stream operators are defined to behave according to the clocking regime. For example, the rule (DFcnstF) ensures the absence of a value when the clock is false. Likewise the unary and binary operators lifted to stream operations $\hat{\diamond}$ and $\hat{\oplus}$ operate only when the argument streams have values present, as in (DFunop) and (DFbinop), and mark absence when the argument streams' values are absent, as shown in (DFunopA) and (DFbinopA).

Note that in the rules (DFmrgT) and (DFmrgF) for when, a value is present on one of the two streams being merged and absent on the other. When a value is absent on the stream corresponding to the boolean variable, values are absent on all streams (DFmrgA). The rules for ite require all streams to have values present, *i.e.*, (DFiteT) and (DFiteF), or all absent, *i.e.*, (DFiteA). We have already discussed the when operation. The fby operation is a bit subtle, and rule (DFfby) may look non-intuitive. However, its formulation corresponds exactly to the Velus formalisation, ensuring that a value from the first argument stream is prepended exactly when a leading value would have been present on the second argument stream. Note that since the streams may be infinite, and the Stream semantics is co-inductive, there is no base case. The operation base-of converts a value stream to a clock, *i.e.*, a boolean stream. The operation respects-clock is formulated corresponding to the Velus definition. The rules (DFtl) and (DFhtl) are obvious.

$$\frac{\mathsf{const}\ bs'\ c = cs'}{\mathsf{const}\ (\mathsf{true}\cdot bs')\ c = \langle c\rangle\cdot cs'}\ (\text{DFcnstT})$$

$$\frac{\mathsf{const}\ bs'\ c = cs'}{\mathsf{const}\ (\mathsf{false}\cdot bs')\ c = \langle\rangle\cdot cs'}\ (\text{DFcnstF})$$

$$\frac{\hat{\diamond}\ es' = os'}{\hat{\diamond}\ (\langle\rangle\cdot es') = \langle\rangle\cdot os'}\ (\text{DFunopA})$$

$$\frac{\hat{\diamond}\ es' = os' \quad v' = \diamond\ v}{\hat{\diamond}\ (\langle v\rangle\cdot es') = \langle v'\rangle\cdot os'}\ (\text{DFunop})$$

$$\frac{es'_1\ \hat{\oplus}\ es'_2 = os'}{(\langle\rangle\cdot es'_1)\hat{\oplus}(\langle\rangle\cdot es'_2) = \langle\rangle\cdot os'}\ (\text{DFbinopA})$$

$$\frac{es'_1\ \hat{\oplus}\ es'_2 = os' \quad v_1\oplus v_2 = v}{(\langle v_1\rangle\cdot es'_1)\hat{\oplus}(\langle v_2\rangle\cdot es'_2) = \langle v\rangle\cdot os'}\ (\text{DFbinop})$$

$$\frac{\mathsf{merge}\ xs'\ ts'\ fs' = os'}{\mathsf{merge}\ (\langle T\rangle\cdot xs')\ (\langle v_t\rangle\cdot ts')\ (\langle\rangle\cdot fs') = \langle v_t\rangle\cdot os'}\ (\text{DFmrgT})$$

$$\frac{\mathsf{merge}\ xs'\ ts'\ fs' = os'}{\mathsf{merge}\ (\langle F\rangle\cdot xs')\ (\langle\rangle\cdot ts')\ (\langle v_f\rangle\cdot fs') = \langle v_f\rangle\cdot os'}\ (\text{DFmrgF})$$

$$\frac{\mathsf{merge}\ xs'\ ts'\ fs' = os'}{\mathsf{merge}\ (\langle\rangle\cdot xs')\ (\langle\rangle\cdot ts')\ (\langle\rangle\cdot fs') = \langle\rangle\cdot os'}\ (\text{DFmrgA})$$

$$\frac{\mathsf{ite}\ es'\ ts'\ fs' = os'}{\mathsf{ite}\ (\langle T\rangle\cdot es')\ (\langle v_t\rangle\cdot ts')\ (\langle v_f\rangle\cdot fs') = \langle v_t\rangle\cdot os'}\ (\text{DFiteT})$$

$$\frac{\mathsf{ite}\ es'\ ts'\ fs' = os'}{\mathsf{ite}\ (\langle F\rangle\cdot es')\ (\langle v_t\rangle\cdot ts')\ (\langle v_f\rangle\cdot fs') = \langle v_f\rangle\cdot os'}\ (\text{DFiteF})$$

$$\frac{\mathsf{ite}\ es'\ ts'\ fs' = os'}{\mathsf{ite}\ (\langle\rangle\cdot es')\ (\langle\rangle\cdot ts')\ (\langle\rangle\cdot fs') = \langle\rangle\cdot os'}\ (\text{DFiteA})$$

$$\frac{\mathsf{when}\ k\ xs'\ es' = os'}{\mathsf{when}\ k\ (\langle k\rangle\cdot xs')\ (\langle v\rangle\cdot es') = \langle v\rangle\cdot os'}\ (\text{DFwhnk})$$

$$\frac{\mathsf{when}\ k\ xs'\ es' = os'}{\mathsf{when}\ k\ (\langle\neg k\rangle\cdot xs')\ (\langle v\rangle\cdot es') = \langle\rangle\cdot os'}\ (\text{DFwhnkA1})$$

$$\frac{\mathsf{when}\ k\ xs'\ es' = os'}{\mathsf{when}\ k\ (\langle\rangle\cdot xs')\ (\langle\rangle\cdot es') = \langle\rangle\cdot os'}\ (\text{DFwhnkA2})$$

$$\frac{\mathsf{fby}\ v\ xs = ys}{\mathsf{fby}\ c\ (\langle v\rangle\cdot xs) = \langle c\rangle\cdot ys}\ (\text{DFfby})$$

$$\frac{\mathsf{fby}\ c\ xs = ys}{\mathsf{fby}\ c\ (\langle\rangle\cdot xs) = \langle\rangle\cdot ys}\ (\text{DFfbyA})$$

$$\frac{\mathsf{respects\text{-}clock}\ H_*\ bs\ vs}{\mathsf{respects\text{-}clock}\ H_*\ (\mathsf{false}\cdot bs)\ (\langle\rangle\cdot vs)}\ (\text{DFresA})$$

$$\frac{\mathsf{respects\text{-}clock}\ H_*\ bs\ vs}{\mathsf{respects\text{-}clock}\ H_*\ (\mathsf{true}\cdot bs)\ (\langle v\rangle\cdot vs)}\ (\text{DFres})$$

$$\frac{\mathsf{base\text{-}of}\ vs = \ bs}{\mathsf{base\text{-}of}\ (v\cdot vs) = \ \mathsf{true}\cdot bs}\ (\text{DFbase1})$$

$$\frac{\mathsf{base\text{-}of}\ vs = \ bs}{\mathsf{base\text{-}of}\ (\langle\rangle\cdot vs) = \ \mathsf{false}\cdot bs}\ (\text{DFbase2})$$

$$\frac{es = v\cdot es'}{(\mathsf{tl}\ es) = es'}\ (\text{DFtl}) \qquad \frac{x\in dom(H_*)}{(\mathsf{htl}\ H_*)(x) = (\mathsf{tl}\ H_*(x))}\ (\text{DFhtl})$$

Fig. 10. Definitions of auxiliary predicates