Towards Refinement Types for Time-Dependent Data-Flow Networks

Jean-Pierre Talpin
INRIA Rennes, France
talpin@irisa.fr

Pierre Jouvelot
MINES ParisTech, PSL Research University, France
pierre.jouvelot@mines-paristech.fr

Sandeep Kumar Shukla
IIT Kanpur, India
sandeeps@cse.iitk.ac.in

Abstract—**The concept of liquid clocks introduced in this paper is a significant step towards a more precise compile-time framework for the analysis of synchronous and polychromous languages. Compiling languages such as Lustre or Signal indeed involves a number of static analyses of programs before they can be synthesized into executable code, e.g., synchronicity class characterization, clock assignment, static scheduling or causality analysis. These analyses are often equivalent to undecidable problems, necessitating abstracting such programs to provide sound yet incomplete analyses. Such abstractions unfortunately often lead to the rejection of programs that could very well be synthesized into deterministic code, provided abstraction refinement steps could be applied for more accurate analysis. To reduce the number of false negatives occurring during the compilation process, we leverage recent advances in type theory – with the definition of decidable classes of value-dependent type systems – and formal verification, linked to the development of efficient SAT/SMT solvers, to provide a type-theoretic approach that considers all the above analyses as type inference problems. To simplify the exposition of our new approach in this paper, we define a refinement type system for a minimalistic, synchronous, stream-processing language to concisely represent, analyze, and verify logical and quantitative properties of programs expressed as stream-processing data-flow networks. Our type system provides a new framework for representing logical time (clocks) and scheduling properties, and to describe their relations with stream values and, possibly, other quantas. We show how to analyze synchronous stream processing programs (à la Lustre, Signal) to enable previously described analyses involved in compiling such programs. We also prove the soundness of our type system and elaborate on the adaptability of this core framework by outlining its extensibility to specific models of computations and other quantas.**

I. Introduction

Designing robust control for cyber-physical systems is a challenging problem for control system engineers. Designing resilient control systems for today's power plants, automotive dynamics, or avionics, are problems that are solved mathematically, simulated in MATLAB/Simulink or similar tools, for validation, and then implemented in software by engineers. The process of going from the mathematical equations to software implementation is often error prone, hence the considerable research over the last two decades on automated code synthesis from formal specifications of control algorithms. Given that control algorithms are equational, to capture the computation involved, data-flow-oriented formal languages such as Kahn networks, Lustre, Signal, Synchronous Data-Flow (SDF) have been proposed in the late 80s [3], [11–13]. Synchronous data-flow-oriented formal languages differ from other data-flow languages in that they are based on the "synchronous hypothesis", which requires that computation be performed as a sequence of reactions to inputs taken from multiple streams of inputs. These input streams are usually either sampled values of physical quantities measured from the physical system under control, or events generated from other parts of the control system such as interrupts, completion signals, acknowledgements, timer signals. One major difference between languages like SDF, Lustre and Signal lies in the underlying model of time. In SDF and Lustre, reactions are totally ordered, and form the necessary artefacts to create the reaction boundaries, namely 'clocks', which have to satisfy certain constraints, checked at compile time. The compilation problem of data-flow syn/poly-chronous languages thus involves a 'clock calculus' which, in the case of Lustre, amounts to checking that all clocks are related to a main one from which the 'activation' of every computation is derived [11]. In the case of Signal, this becomes the more general issue of finding, first, if such a main clock exists [2], and, second, how others relate to it. This problem is solved, in most practical cases, by abstract reasoning on Boolean relations. Additionally, Satisfiability (SAT) Modulo Theory (SMT) solver-based extensions have recently been kludged into the implementation of these calculi and synthesis engines [9], [21].

This entire process of static analysis, capturing program properties in a logic with suitable decision procedures, is the crux of extending synthesis techniques to quantitative specifications. In this paper, instead of integrating time and quantitative reasoning at the implementation level of the synthesis engine or defining specific type systems to represent each of its logical, quantitative or causal aspects [4], [8], [18], we propose a generic type theory based on refinement types to integrate this ability into the language's static semantic itself and improve synthesizability of programs from its data-flow specifications [9], [21]. The advantage of doing that is manifold, but, most importantly, the type inference algorithms themselves will also decide synthesizability.

We adopt the liquid type theory introduced by Jhala et al. [25], [30–32] to apply and extend it to the context of timed data-flow languages by the introduction of properties on time and causality: the theory of what we call 'liquid clocks'. Capturing quantitative properties of timed data-flow specifications, in a decidable manner, using liquid types, opens to a variety of applications, from the integration of contract systems, the traceability of program properties from specification to generated code, to translation validation and certified code generation. Moreover, liquid types allow to revisit many of the ad-hoc and problem-specific algebras and/or type theories that have been proposed to capture many variations of the strictly synchronous hypothesis using periodic, multi-rate, affine, regular, integer, cyclo-static, continuous time models... all into one single, unified, verification framework. Liquid types also open to considering a large variety of models of computation and communication (MoCC), not only synchronous and polychronous data-flow in the spirit of SDF, Simulink, Lustre and Signal, but also multi-rate, cyclo-static, data-parallel MoCC.

Outline Section II starts with the presentation of a generic data-flow language, with its typing system, to describe the network structuring cooperating stream functions. Section III presents our refinement type inference system. A constructive dynamic semantics of the language is given in Section IV for the purpose of stating a subject reduction property, Section V. Sections VI and VII extend our data-flow language with primitives that implement the MoCC of synchronous data-flow languages like Lustre and Signal; we tackle the properties of determinism and deadlock-freedom by means of an interpretation of liquid types. Section VIII addresses the related work before the conclusion.

II. A simple data-flow language

We introduce as data-flow language a lambda-calculus of expressions over stream functions; even though minimalistic, it is expressive enough to handle discrete streams of timed events

Syntax An expression e defines here a network of stream functions build from definitions, d; one can reference a stream x, apply it to a function expression, in $e\,x$, or add a local definition d in the scope of e with $\mathsf{let}\,d\,\mathsf{in}\,e$. A definition d is either a (non-recursive) function $f(x)=e$ that parameterizes the expression e over x, or the simultaneous composition of (possibly recursive) equations $x=e$. An equation $x = y \star z$ defines the stream x by the repeated application of the operator $\star$ on values incoming from y and z. It usually requires the availability of a value along y and z before performing the operation that defines x. Such an occurrence is called an event of x and its repetition, a clock, $\hat{x}$.

$$\begin{array}{lll@{\quad}lll} e ::= & x & \textit{stream} & d ::= & f(x)=e & \textit{function} \\ & \mid e\,x & \textit{application} & & \mid x=e & \textit{equation} \\ & \mid \mathsf{let}\,d\,\mathsf{in}\,e & \textit{definition} & & \mid d \,\|\, x=e & \textit{composition} \end{array}$$

Meta-variables used in the grammars and rules follow some naming conventions. Streams are noted x,y,z and sometimes c,n if they hold a constant value (a Boolean, an integer). Operators, seen as stream functions, or processes are written f. The constant identifier $()$ stands for void.

Static semantics The type system for stream functions defines three classes of types for basic data, noted b, streams, noted s, and program objects, noted t. A stream type s is a data-dependent type structure consisting of a value identifier v, a base type b and a property p of v; the stream value variable v denotes the output of primitive operators on streams. Streams are first-order objects (they do not carry functions). A *liquid type* t is either a stream s or a function $x:s \to t$. The liquid type $x:s \to t$ of a function associates a property to the type s of its parameter x and its result (a program object t). For instance, the type $x{:}\mathsf{int} \to \langle y : \mathsf{int} \mid y \geqslant x\rangle \to \langle v{:}\mathsf{int} \mid v \geqslant y\rangle$ denotes a function that maps an integer x to a function that, first, expects its argument y to be greater or equal to x and, second, yields a result greater than y. Finally, a type environment E is a set registering declared stream and function names with their types. We note $E,(x:t)$ for the extension of E with a name x of type t; $E,(x:t)$ is defined iff $x \notin dom(E)$. The empty environment $[]$ can be omitted. In [29], stream functions also support pairs (x,e), of type $s \times t$, and data-type polymorphism, noted $\Lambda\alpha.e$, of type $\forall\alpha.t$, instantiated to $t[b/\alpha]$ by $e[b]$, as in [25].

$$\begin{array}{lll@{\quad}lll} b ::= & () \mid \mathsf{bool} \mid \mathsf{int} & \textit{data-type} & t ::= & s \mid x{:}s \to t & \textit{type} \\ s ::= & \langle v{:}b \mid p\rangle & \textit{stream-type} & E ::= & [] \mid E,(x:t) & \textit{context} \end{array}$$

Logical Qualifiers Properties p and atomic logical qualifiers q are defined on the quantifier-free logic of uninterpreted functions and linear arithmetic (QF-EUFLIA) amenable to automatic verification using, e.g., SMT solving or theorem proving [20]. Properties are limited to the conjunction of and implication (or equivalence) between qualifiers q, in $\boldsymbol{Q}$. A liquid type always holds a property p (of type boolean) in conjunctive form. In the same manner as [32], qualifiers q are typed boolean formulas constructed by a grammar starting from constants b (Booleans true and false), streams x, and unary and binary measures $\star$. Measures $\star$ are uninterpreted boolean and integer operators. The clock operator $\hat{x}$ stands for the symbolic period of data on a stream x; it is a boolean here, but could be an integer, depending on the model of time under consideration. Another peculiarity is the distinction between causal (non-reflexive) input-output stream equality, noted $x=q$, and its implied reflexive (a-causal) logical equality $x == q$ and equivalence $x \Leftrightarrow q$ (see Section VII).

$$\begin{array}{lll} p & ::= & & \textit{liquid refinement} \\ & \mid & q & \textit{qualifier} \\ & \mid & p \wedge p & \textit{conjunction} \\ & \mid & p \Rightarrow p & \textit{implication} \\ q & ::= & v \mid \star q \mid q \star q & \textit{qualifier, in } \boldsymbol{Q} \\ \star & ::= & & \textit{measure} \\ & \mid & \hat{} & \textit{clock} \\ & \mid & \wedge \mid \neg \mid \Rightarrow \mid \vee \mid - \mid & \textit{boolean} \\ & \mid & \times \mid + \mid - \mid < \mid \leqslant \mid = \mid == & \textit{integer} \end{array}$$

III. Type inference

The proposed type inference system is defined in the spirit of Razou et al. [32] and extends the specification, inference and verification of timed, or clocked, properties. We do not need to use polymorphism at all since all process types are data parameter-dependent; defining the appropriate model and type inference algorithm becomes this way a lot easier, while providing little limitations for our application domain.

Intrinsic stream functions The typing judgment $E \vdash \star : t$ for intrinsic functions $\star$ is very much in the spirit of related work in synchronous programming as to its logical, timed and arithmetic properties [5], [9], [25]. In the remainder, we use the following abbreviations to designate liquid types in which a value variable v isn't used or when its scope is not ambiguous.

$$b \triangleq v{:}b \triangleq \langle v{:}b \mid \mathsf{true}\rangle \qquad b\langle p\rangle \triangleq \langle v{:}b \mid p\rangle$$

An intrinsic synchronous expression/definition $x = y \star z$ requires its input and output streams x,y,z to be present at the same time: it repeatedly pulls values from input streams, computes a result and pushes it on the output stream. This yields two timing invariants. First, the clocks of x,y,z must be true at the same time: $\hat{x} \Leftrightarrow \hat{y} \Leftrightarrow \hat{z}$. Second, the value of the output stream is defined only if its clock is present: $\hat{x} \Rightarrow x = y \star z$. Also, Boolean and integer constants are defined by constant streams.

$$\begin{array}{ll} E \vdash c & : \langle v{:}\mathsf{bool} \mid \hat{v} \Rightarrow (v = c)\rangle \\ E \vdash \mathsf{not} & : x{:}\mathsf{bool} \to \langle v{:}\mathsf{bool} \mid (\hat{x} \Leftrightarrow \hat{v}) \wedge \hat{v} \Rightarrow (v = \neg x)\rangle \\ E \vdash \mathsf{plus} & : x{:}\mathsf{int} \to y{:}\mathsf{int} \to \langle v{:}\mathsf{int} \mid (\hat{x} \Leftrightarrow \hat{y} \Leftrightarrow \hat{v}) \wedge \hat{v} \Rightarrow (v = x + y)\rangle \\ E \vdash \mathsf{pos} & : x{:}\mathsf{int} \to \langle v{:}\mathsf{bool} \mid (\hat{v} \Leftrightarrow \hat{x}) \wedge \hat{v} \Rightarrow (v = (x \geqslant 0))\rangle \end{array}$$

Well-formed types In a refinement type system, types are subject to the well-formed relation $E \vdash t\checkmark$. First, it ensures the type consistency of properties generated during type inference. For instance, the rule for stream types $\langle v{:}b \mid p\rangle$ applies the type inference rules to check that the property has a Boolean type. Second, it ensures proper lexical scoping of terms during type inference. By using it, the abstraction of qualifiers is strictly enforced once a term escapes the scope of its definition. The property of well-formedness extends to type environments E.

$$E \vdash b\,\checkmark \qquad \frac{E, v{:}b \vdash p{:}\mathsf{bool}}{E \vdash \langle v{:}b \mid p\rangle\,\checkmark}$$

$$\frac{E \vdash s\,\checkmark \quad E, x{:}s \vdash t\,\checkmark}{E \vdash (x{:}s) \to t\,\checkmark} \qquad \frac{E \vdash t\,\checkmark \quad E, x{:}t \vdash F\,\checkmark}{E \vdash x{:}t, F\,\checkmark}$$

In the rule for stream types $\langle v{:}b \mid p\rangle$, we assume that p is well-typed with $E, v{:}b \vdash p{:}\mathsf{bool}$. This means that we interpret p as a Boolean expression e and check that it does not contain any variable free in the scope of E and v, and that the operations it includes are of correct arity and type.

Sub-typing The abstraction, or reduction, of inferred properties is performed via sub-typing, as for the liquid types defined in [32], to yield a type system decidable in QF-EUFLIA. The sub-typing rule for streams, Rule (S-SIG), makes use of the logical interpretation $(\!|p|\!)$ of a property p (see below) to convey the expected definition that $E \vdash s \preceq t$ whenever $(\!|E|\!) \Rightarrow (\!|s|\!) \Rightarrow (\!|t|\!)$ is a tautology. As in [32], it is interpreted as a formula in QF-EUFLIA, in order to facilitate SAT/SMT checking ($\models q$ indicates that q is a tautology).

$$E \vdash b \preceq b \quad \text{(S-BAS)} \qquad \frac{\models (\!|E, v{:}b|\!) \Rightarrow (\!|p|\!) \Rightarrow (\!|p'|\!)}{E \vdash \langle v{:}b \mid p\rangle \preceq \langle v{:}b \mid p'\rangle} \quad \text{(S-SIG)}$$

A logical interpretation $(\!|t|\!)$ is defined by induction on the structure of types and environments as the interpreted conjunction of properties implied by the meaning of a type t. We write $p[x/v]$ for the substitution of v by x in p; tt denotes the true value.

$$(\!|\langle v : b \mid p\rangle|\!) = p \qquad (\!|x{:}s \to t|\!) = \mathit{tt}$$
$$(\!|x{:}\langle v : b \mid p\rangle|\!) = p[x/v] \qquad (\!|E, x{:}t|\!) = (\!|E|\!) \wedge (\!|t|\!)$$

Well-formedness and sub-typing allow to define the restriction E_x of a stream x from an environment $E = F, x{:}s$ by $\vdash E_x\,\checkmark$ where $(\!|E_x|\!)$ is the maximal implicant such that $E \vdash F \preceq E_x$ (sub-typing is extended to environments).

Inference system The type inference system inductively defines the relation $E \vdash e : t$ on expressions. Its co-inductive relation $E \vdash d{:}F$ associates a definition d with an environment F under the assumptions E. Rule (T-SIG) defines a value stream v from a reference to the named stream x, binding the clock and value of v and x together. Sub-typing is allowed at any time during type inference, by using Rule (T-SUB), but requires the production of a well-formed type t' with respect to the parent environment E. Beta-reduction expressions for application and scoping operate differently. Rule (T-APP) defines the type of the application $e\,y$ of a stream y to a function e by returning its result type t, and by substituting the name of its formal parameter x by that of the actual, y, noted $t[y/x]$. Streams x and y must have the same type s. Rule (T-LET) handles the local definition of d in e. First, the type of d must be inferred. It is an environment F listing the type of all names defined by d. This environment is then used with E to determine the type t of e and return it. However, since t must escape the scope of d, we have to ensure that t does not reference a name introduced in F, hence check that it is well-formed with respect to E. The rule for definition uses the type inference relation $E \vdash e{:}t$ to associate names x defined in equations $x{=}e$ to the corresponding type environment $(x{:}s)$, in Rule (T-DEF). Rule (T-COM) composes them. Rule (T-ABS) is that of lambda-abstraction. It chooses a type $b\langle p\rangle$ for the formal parameter x and deduces the type t of its body e. The resulting function type must be well-formed with respect to E.

$$E, f{:}t \vdash f{:}t \quad \text{(T-FUN)}$$

$$E, x{:}\langle v{:}b \mid p\rangle \vdash x{:}\langle v{:}b \mid p \wedge \hat{x} \Leftrightarrow \hat{v} \Rightarrow v = x\rangle \quad \text{(T-SIG)}$$

$$\frac{E \vdash e{:}t \quad E \vdash t \preceq t' \quad E \vdash t'\,\checkmark}{E \vdash e{:}t'} \quad \text{(T-SUB)}$$

$$\frac{E \vdash e : (x{:}s) \to t \quad E \vdash y : s}{E \vdash e\,y : t[y/x]} \quad \text{(T-APP)}$$

$$\frac{E, F \vdash d{:}F \quad E \vdash F\,\checkmark \quad E, F \vdash e{:}t \quad E \vdash t\,\checkmark}{E \vdash \mathsf{let}\ d\,\mathsf{in}\,e{:}t} \quad \text{(T-LET)}$$

$$\frac{E \vdash e : s}{E \vdash x{=}e : (x{:}s)} \quad \text{(T-DEF)}$$

$$\frac{E \vdash d : F \quad E \vdash x{=}e : F'}{E \vdash d \,|\, x{=}e{:}F, F'} \quad \text{(T-COM)}$$

$$\frac{E \vdash x{:}b\langle p\rangle\,\checkmark \quad E, x{:}b\langle p\rangle \vdash e{:}t}{E \vdash f(x{:}b){=}e{:}(f{:}(x{:}b\langle p\rangle) \to t)} \quad \text{(T-ABS)}$$

IV. Constructive semantics

We consider the constructive, small-step, reduction semantics of [28] to describe the behaviour of data-flow networks. Its key feature is to embed a set of stream statuses $\mathbb{D} = \mathbb{V} \cup \{?, \bot, \top, \lightning\}$ into a lattice of data values $\mathbb{V} = \mathbb{B} \cup \mathbb{Z}$. Status ? stands for unknown, $\bot$ for absent or inhibited, $\top$ for present or activated, and $\lightning$ for inconsistent. We write $\mathbb{V}^{\bot} = \mathbb{V} \cup \{\bot\}$ for the set of final stream statuses. Starting from $\mathbb{D}$, we define a partial order $\sqsubseteq$ on $\mathbb{D} \times \mathbb{D}$: the *greater* a status is, the more information we have about it. In [28], Status $\lightning$ is the greatest element. For all $v \in \mathbb{V}$, one has the relations $? \sqsubseteq \bot \sqsubseteq \lightning$ and $? \sqsubseteq \top \sqsubseteq v \sqsubseteq \lightning$.

Without loss of generality, the language of stream functions of Section II can be (syntactically) reduced, via a rewriting $e \rightsquigarrow d$ (see [29]), to the scoped composition of synchronous simultaneous data-flow equations, seen as base definitions d.

$$d ::= x = y \star z \mid d|d \mid d/x \qquad \textit{base definitions}$$

In the remainder, base definitions are defined by the composition $d|d$ of equations $x = y \star z$ built from intrinsic and data-flow operators $\star$ in $\{\mathsf{and}, \mathsf{or}, \mathsf{not}, \ldots\}$. The scope of a stream x is lexically bound to a definition d by d/x, as in process calculi. We note $in(d)$ and $out(d)$ the input and output streams in d. $fv(d)$ is the unbound streams of d, a notation extended to expressions and environments.

The evaluation of a definition d is then defined by a monotonic progress rule [29] $r, d \rightharpoonup r', d'$ that defines all legal transitions that gain knowledge from r, d by evolving into r', d'. A registry r is a function mapping a finite set of stream names x to their statuses in $\mathbb{D}$. The relation $\rightharpoonup$ iteratively gains knowledge about the status of the stream variables defined in its domain $dom(r)$.

Example Every time the execution of the `countdown` program

$$c = o\,\mathsf{pre}\,0 \mid o = n\,\mathsf{default}\,(c-1) \mid y = \mathsf{true}\,\mathsf{when}\,(c=0) \mid n\,\mathsf{sync}\,y$$

is triggered, it provides the value of a decremented count, in c, on its output stream o. If that count reaches 0, a condition tested by y, the output stream o synchronizes with the input stream n to reset the counter. The execution of `countdown` is depicted by a series of steps (see Section VI for the semantics of stream operators). Changes are marked with a bullet •. The previous value x of the counter c is noted $count_x$. Let us assume for now that the environment has triggered execution `countdown` by furnishing the input stream n with the value 1 to start counting. Consequently, the initial count 0 can be loaded into c and the new 1 is stored in place of it. Once c is known, y can be deduced. We are left with the synchronization constraint $n \,\mathsf{sync}\, y$ which, luckily, is true, as both n and y are present.

$$\begin{array}{llllll}
 & (c,?) & (n,1_\bullet) & (o,?) & (y,?) & \mathit{count}_0 \\
\rightharpoonup & (c,?) & (n,1) & (o,1_\bullet) & (y,?) & \mathit{count}_0 \ \ \mathit{(from}\ \mathsf{default}\mathit{)} \\
\rightharpoonup & (c,0_\bullet) & (n,1) & (o,1) & (y,?) & \mathit{count}_{1_\bullet} \ \ \mathit{(from}\ \mathsf{pre}\mathit{)} \\
\rightharpoonup & (c,0) & (n,1) & (o,1) & (y,\mathit{tt}_\bullet) & \mathit{count}_1 \ \ \mathit{(from}\ \mathsf{when}\mathit{)} \\
\rightharpoonup & (c,0) & (n,1) & (o,1) & (y,\mathit{tt}) & \mathit{count}_1 \ \ \mathit{(from}\ \mathsf{sync}\mathit{)}
\end{array}$$

V. Soundness of liquid clocks

The status of a given stream x at all times is obtained by its clock $\hat{x}$. When looking at boolean models for clocks, we posit, by construction, that $\hat{x}$ is true at a given point in time (a time tag) iff x holds a value at the same relative stream point, or x is activated and holds $x = \top$. Conversely, $\hat{x}$ is false iff x does not carry a value (with respect to another stream's time tag), denoted $x = \bot$; hence $\hat{x} \Leftrightarrow x \neq \bot$. In the remainder, all properties pertaining to values are guarded or conditioned by properties on clocks (e.g., $\hat{x} \Leftrightarrow v = x$) to enforce stratified reasoning on boolean expressions.

A registry r can be seen as a partial or complete (when its statuses are values or absence, i.e. $r(x) \in \mathbb{V}^\bot$ for all x in $dom(r)$) model of properties p specified in an environment E. A partial or incomplete model maps to at least one unknown or undetermined (present) status. Additionally, we say that a model r' progresses from r, written $r \sqsubseteq r'$, iff $r(x) \sqsubseteq r'(x)$ for all $x \in dom(r) = dom(r')$. We write $r \models (\!|p|\!)$ to mean that the model r satisfies the logical meaning of Property p.

Definition 1 (Model of registry): We say that E is a type environment for r, i.e. $\vdash r : E$, iff $\vdash E \checkmark$, $dom(E) = dom(r)$ and, for all $x \in dom(r)$, $E(x) = \langle v\!:\!b \mid p\rangle$ and $\vdash r(x) : b$. In particular, $\vdash ?\!:\!b$, $\vdash \bot\!:\!b$, $\vdash \top\!:\!b$ for all b. A complete registry r is a model of E, i.e. $r \models E$ iff $\vdash r : E$ and, for all $x \in dom(r)$, $E(x) = \langle v\!:\!b \mid p\rangle$ and $r \models (\!|p[x/v]|\!)$. A partial registry r is a model of E iff there exists a complete model $r' \sqsupseteq r$ s.t. $r' \models E$.

Subject reduction and the corresponding type preservation property consider an expression e of type t under well-formed hypothesis E and reduces it to the definitions $x = e \rightsquigarrow d$ using a fresh $x \notin fv(e)$. If, given a model $r \models E, (x : t)$, evaluation progresses by $r, d \rightharpoonup r', d'$, then $r' \models E, (x, t)$ and $E, (x, t) \vdash d' : (x, t)$ (see proof in [29]).

Theorem 1 (Type preservation): Let $E \vdash e : t$, $x \notin dom(E)$, $F = E, x : t$ and $x = e \rightsquigarrow d$. If $r \models F$ and $r, d \rightharpoonup r', d'$, then $r' \models F$ and $F \vdash d' : (x\!:\!t)$.

VI. Typing clock operators

Synchronous data-flow languages like SDF, Lustre and Signal support similar primitives to delay, merge and sample streams.

Delay In a synchronous data-flow language, a delay equation $x = y \,\mathsf{pre}\, z$ (in Lustre, `x = z -> pre y`; in Signal `x := y$1 init z`) sends the value of z (a constant c) along the output stream x and stores the value w of y in place of z. As a result, it delays the delivery of events from stream y by one evaluation tick and prefixes this output by z. To determine the type of pre in lieu of (T-FUN) with $f = \mathsf{pre}$, some temporal reasoning is in order. pre initially (at clock i_0) accepts an input constant z and then (at clock $\neg i_0$) outputs the previous value of y (at some clock i_{n-1}) to the output stream v (at clock i_n), $n > 0$. Therefore, the liquid type t_z of z (at the first instant i_0 of v) and the liquid type t_y of y (at instants $\neg i_0$ of y) need to have a common implicant that mentions none of the branches or fictive clocks $i_0 \ldots i_n$. Let p be that common upper-bound invariant of x, i.e., the property guaranteed by x at all times. One gets:

$$E \vdash \mathsf{pre} : x\!:\!b\langle p\rangle \rightarrow y\!:\!b\langle p\rangle \rightarrow b\langle v\!:\!b \mid p \wedge \hat{v} \Leftrightarrow \hat{x} \Leftrightarrow \hat{y}\rangle$$

Merge Lustre and Signal only differ in the way merge (and sampling) operations are processed in time. This, in turn, makes an important difference as to how and when clocks can be computed. In Signal, an equation `x := y default z` defines the output x by y if it is present and by z otherwise. In Lustre, merge is implemented by the conditional `x = if c then y else z` over four synchronous streams x, c, y, z. At all times, x takes the value of y if c is true and takes z otherwise. Path sensitivity reasoning is used to define the axioms for default and if by considering the clocks of input streams. If we apply the same reasoning to Lustre's conditional, we get value relations guarded by clocks.

$$E \vdash \mathsf{if}: c\!:\!\mathsf{bool} \rightarrow y\!:\!b \rightarrow z\!:\!b \rightarrow \langle v\!:\!b \mid \hat{v} \Leftrightarrow \hat{c} \Leftrightarrow \hat{y} \Leftrightarrow \hat{z} \ \wedge \ \begin{array}{l} c \Rightarrow v = y \\ \wedge \neg c \Rightarrow v = z\rangle \end{array}$$

For Signal's default axiom, clocks are these of both input streams (data-flow paths). The output v is defined by the stream y when its clock is present and by the stream z otherwise (at the clock $\hat{z}$ "minus" $\hat{y}$). This analysis yields the conjunction of properties that defines the type of default.

$$E \vdash \mathsf{default}\!: x\!:\!b \rightarrow y\!:\!b \rightarrow \langle v\!:\!b \mid \hat{v} \Leftrightarrow \hat{x} \vee \hat{y} \wedge \hat{x} \Rightarrow v = x \wedge \hat{y} - \hat{x} \Rightarrow v = y\rangle$$

Sample The same reasoning applies to the rule for sampling: in Lustre, `x = y when z` and, in Signal, `x := y when z`. In Lustre, y, z are synchronous and x is defined by y when z is true. In Signal, the output stream is defined by y if z is true.

$$E \vdash \mathsf{when}_{lu} : y\!:\!b \rightarrow z\!:\!\mathsf{bool} \rightarrow \langle v\!:\!b \mid \hat{v} \Leftrightarrow \hat{y} \Leftrightarrow \hat{z} \wedge z \Rightarrow v = y\rangle$$
$$E \vdash \mathsf{when}_{sig} : y\!:\!b \rightarrow z\!:\!\mathsf{bool} \rightarrow \langle v\!:\!b \mid \hat{v} \Leftrightarrow (\hat{y} \wedge \hat{z} \wedge z) \wedge \hat{v} \Rightarrow v = y\rangle$$

VII. Safety Properties

Critical safety properties are these guaranteeing the correct executability of programs synthesised from data-flow specifications. Dead-lock freedom and schedulability can be deduced from liquid types and represented using specific algebraic structures used e.g., in the Signal compiler, to perform whole-program analysis.

Schedulability The determination of a static schedule of operations is an essential part of the compilation of synchronous languages. Imperative synchronous languages such as Esterel and SyncCharts define graphs representing write-to-read relations between program instructions [1], [15]. Data-flow synchronous languages define graphs to materialize causal dependencies from definitions to uses of stream values in programs [17], [23]. Liquid types allow us to formulate schedule synthesis from the type of a data-flow network. This is done by constructing a guarded scheduling graph (in the form of [21]) from the interpretation of uninterpreted guarded equations present in liquid types. We implement this by the introduction of additional qualifiers q of the form $\hat{y} \Rightarrow x \rightarrow z$, that abstract (or, equivalently, are implied by) inferred properties of the form $\hat{y} \Rightarrow z = q[x]$, where $q[]$ is a 'context', or term with a hole in it: $q[] ::= [] \mid \star q[] \mid q[] \star q \mid q \star q[]$.

$$q ::= \ldots \mid q \Rightarrow a \rightarrow a \quad \textit{abstract qualifiers}$$

where $a ::= v \mid \hat{v}$. The type of a synchronous data-flow network features properties of two forms: relations between clocks and/or boolean conditions as well as (directed) equalities guarded by clocks $\hat{x} \Rightarrow y = q[z]$. One can interpret the latter as causal relations to build a graph of scheduling relations of the form

$z \rightarrow y$ to mean that the computation of z precedes that of y at some Clock $\hat{x}$. However, stream equality $x = y$ is causal and needs to be disambiguated from logical equality, written $x == y$, by the decomposition of an equation $x = y$ into its implied logical equality $x == y$ and causal relation: $y \rightarrow x$ and $\hat{x} \rightarrow x$. Additionally, a clock $\hat{v}$ defined by a boolean condition x depends on the value of the stream x, i.e., $\hat{v} \Leftrightarrow x$. Therefore, it implies a causality relation $x \rightarrow \hat{v}$, as the clock of v cannot be determined before x is computed, along Clock $\hat{x}$. Last, and thanks to the sub-typing rule, the introduction (and elimination) of these relations can be performed in any place suited, e.g. to abstract a local stream x in a definition d from its type by transitivity. The scheduling relation induced by E, noted $\rightarrow_E$, is recursively defined as:

- for all $x \in dom(E)$, $\hat{x} \Rightarrow \hat{x} \rightarrow_E x$;
- if $(\!|E|\!) \models (\!|a \Rightarrow x = q[y]|\!)$, then $a \Rightarrow y \rightarrow_E x$;
- if $(\!|E|\!) \models (\!|\hat{x} \Leftrightarrow q[y]|\!)$ for some context q, then $y \rightarrow_E \hat{x}$;
- if $q \Rightarrow a \rightarrow_E a'$ and $q' \Rightarrow a' \rightarrow_E a''$, then $(q \wedge q') \Rightarrow a \rightarrow_E a''$.

Scheduling graphs are subject to a set-theoretic containment relation $\subseteq$. Now, since our liquid type system permits property elimination using the sub-typing rule, not all types of a given data-flow network are good candidates for causal analysis. In fact, the only equivalence class of interest is that maximal with respect to $\subseteq$. We say that E is *d-maximal* iff $E \vdash d : E$ and, for all $F \vdash d : F$, one has $\rightarrow_F \subseteq \rightarrow_E$.

Definition 2 (Schedulability): Let $E \vdash d : E$ such that E is d-maximal. d is schedulable, or deadlock-free, iff, for all $x \in dom(E)$, $q \Rightarrow x \rightarrow_E x$ invalidates q, i.e., one has $(\!|E|\!) \models \neg(\!|q|\!)$.

Example The countdown function of the previous section, and its type specification, yields the following causal stream relations.

$$\begin{array}{lll} c & = o \text{ pre } 0 & \\ \mid o & = n \text{ default } x & \hat{n} \;\Rightarrow\; n \rightarrow o \\ \mid x & = c - 1 & \hat{x} - \hat{n} \;\Rightarrow\; x \rightarrow o \\ \mid y & = \text{true when } (c = 0) & \hat{c} \;\Rightarrow\; c \rightarrow x \\ \mid () & = n \text{ sync } y & \hat{c} \;\Rightarrow\; c \rightarrow \hat{y} \end{array}$$

Patience The correctness of a synchronous data-flow network also relies on a time-dependent notion of determinism: a latency-insensitive circuit is called patient [7]. Similarly, a synchronous program is said *endochronous* iff it is able to autonomously decide when its streams need to be read or written, i.e. when their status is present or absent. In Lustre (or Simulink or SDF), programs are endochronous stream functions, since all input streams of a function are, by construction, synchronized.

In Signal, endochrony is defined with respect to the asynchronous stream interface of synchronous programs [28], i.e., their capability to deterministically peek values from input streams based on their specification. This internal, or implicit, computation of the program is synthesized by the compiler from a reasoning on clocks called hierarchization, which builds a dominance relation between clocks to sort those that can be computed from others. The strict dominance relation $x \gg_E y$ means that $\hat{y}$ can be computed or deduced from $\hat{x}$ in E, while the equivalence relation $x \sim_E y$ means that $\hat{x}$ and $\hat{y}$ can be deduced from each other:

- if $(\!|E|\!) \models (\!|\hat{x} \Leftrightarrow \hat{y}|\!)$, then $x \sim_E y$;
- if $(\!|E|\!) \models (\!|\hat{x} \Leftrightarrow y|\!)$, then $y \gg_E x$.

The hierarchy of E, noted $\gg_E^*$, is the closure of the dominance and synchrony relations $\gg_E$ and $\sim_E$: if $(\!|E|\!) \models (\!|\hat{x} \Leftrightarrow \hat{y} \star \hat{z}|\!)$ and $y \ll_E^* w \gg_E^* z$ for some operator $\star$ and stream w, then $w \gg_E^* x$. It is used to formally define the concept of patience. A well-formed tree defines the control-flow of a well-synchronized program.

Definition 3 (Patience): Let $E \vdash d : E$ where E is d-maximal. d is patient iff $\gg_E^*$ is a well-formed tree, i.e.:

- there exists $x \in dom(E)$ s.t. $x \gg_E^* y$ for all $y \in dom(E)$;
- if $x \gg_E^* y \ll_E^* z$, then $x \gg_E^* z$ or $z \gg_E^* x$.

Example For the countdown function, typed in Environment E, one syntactically obtains the following clock relations.

$$\begin{array}{llccc} c & = o \text{ pre } 0 & c & \sim_E & o \\ \mid o & = n \text{ default } x & & & \\ \mid x & = c - 1 & c & \sim_E & x \\ \mid y & = \text{true when } (c = 0) & c & \gg_E & y \\ \mid () & = n \text{ sync } y & y & \sim_E & n \end{array}$$

Execution can thus be triggered by a stream of the $\gg_E$-highest class $\{o, c, x\}$, here obviously o, by setting its status to present. The status of all other streams can be deduced from that of o.

Progress with patience The above definitions are used to specify the notion of progress in the framework of liquid types: if d is schedulable and patient, then it progresses. In Lustre, and similarly Simulink and Ptolemy's SDF, all the input streams of a block are synchronous, by construction. This implies that the $\sim_E$-equivalence class of a program in its d-maximal environment E contains all its input streams, yielding progress.

Proposition 1 (Synchronous progress): Let $E \vdash d : E$, where E is d-maximal while d is a schedulable and patient program. Let $\vdash r : E$ such that, for all $x \in in(d)$, one has $r(x) \in \mathbb{V}^\perp$ and $r(x) = ?$ otherwise. Then, $r, d \rightharpoonup^* r', d'$ and, for all $x \in out(d)$, one has $r'(x) \in V^\perp$.

Discussion The above analysis uses the causality graphs and dominance structures of the Signal compiler to perform whole-program analysis, which is done by instantiating all locally defined streams x in scoped definitions d/x (Skolemization). A modular extension of such an analysis naturally requires to scope these local streams when necessary (e.g., to define local clocks) by regarding them as ressources [16], or by using existential quantifiers ([27] for Signal, [9] for SAT/SMT verification).

VIII. Related Work

The framework of liquid clocks introduced in this paper relates to the theory of refinement types developed by the Liquid Haskell project of Jahla et al. [30]. The type system expresses quantitative properties on Boolean and integer values and indexes in the spirit of [25] and the inference system of [32]. Timed properties are represented by guarded proposition much like Pnueli's synchronous transition systems (STS) [24] which makes them amenable to SAT/SMT-verification related works [9], [21], here, by expressing them in QF-EUFLIA logic. Merging both approaches offers a powerful logical framework for both analysis, verification and code generation. It allows us to revisit and improve several lines of earlier works concerned with clock calculi [2], [14], [22], scheduling analysis [6], [17], [23], [26], verification by abstract interpretation [5], [9] or just type-based analysis [4], [8], [18], [27], and yet extend and apply these to a context now far more general than synchronous data-flow, with the correct theoretical concepts to guarantee type soundness. Most related data-flow synchronous languages [3], [8], [10], [11], [13], [18] now have well-developed and understood analysis and verification frameworks to help validate specification correctness and automatically generate code. While many different approaches have been considered to cast these analyses and verification techniques into the framework of type theory, none have been prominently successful in practice. An obvious choice seems to have leaned toward type polymorphism in early attempts [8], [27], albeit parametric polymorphism appears in practice inadequate to represent value dependencies: it makes type inference prone to variable capture, which can only be circumvented by over-approximations and results in inefficiency. Another possible choice is to represent clocks using regular expressions [10], [18], which works well with sub-typing, but unfortunately builds types of exponential space complexity (in the size of the network under analysis), which polymorphic type inference applied to scheduling also does [4], [27]. One hence seeks drastic approximations (envelopes, counters, harmonics, adapters) or sophisticated constraint simplifications to reduce complexity. As demonstrated by Kloos et al. [16] for asynchronous process calculi, in [19] for TLA+, refinement types offer a valuable alternative to the above from several standpoints, both in terms of soundness guarantees, space complexity and abstraction capabilities. Sub-typing in refinement type systems defines a sound and effective means of abstraction, very much comparable, and actually implemented with, widening techniques as in abstract interpretation and reduction techniques as in model checking.

IX. Conclusions

We enriched the refinement type framework of liquid types with liquid clocks to capture quantitative properties of data-flow specifications in a decidable, concise and sound manner. We defined a simple data-flow language and equipped it with a static semantics founded on these liquid types and clocks, seen as property expressions defined over a rich algebra amenable to automatic verification. We showed that such a system is sound with respect to the dynamic small-step reduction semantics of a core version of the data-flow language. We illustrated the use of our new framework for the typing of clock-related primitives and for the specification of safety properties such as schedulability or patience. We are currently furthering our experiments with an evolving prototype that uses Liquid Haskell as language front-end and Z3 as SAT/SMT verifier, opening up to unprecedented expression capabilities.

Acknowledgments This work is partly funded by the ANR project FEEVER, and by the USAF Office for Scientific Research, grant FA8655-13-1-3049. The authors thank Imré Frotier de la Messelière for comments on a previous version of this paper.

References

[1] J. Aguado, M. Mendler and R. von Hanxleden and I. Fuhrmann. "Grounding Synchronous Deterministic Concurrency in Sequential Programming". European Symposium on Programming, LNCS v. 8410. Springer, 2014.

[2] P. Amagbegnon, L. Besnard, and P. Le Guernic. "Implementation of the data-flow synchronous language SIGNAL". Conference on Programming Language Design and Implementation. ACM, 1995.

[3] A. Benveniste, P. Le Guernic, C. Jacquemot. "Synchronous programming with events and relations: the SIGNAL language and its semantics". Science of Computer Programming. Elsevier, 1991.

[4] A. Benveniste, T. Bourke, B. Caillaud, B. Pagano, and M. Pouzet. "A Type-based Analysis of Causality Loops in Hybrid Systems Modelers". In International Conference on Hybrid Systems. ACM, 2014.

[5] F. Besson, T. Jensen, and J.-P. Talpin. "Polyhedral analysis for synchronous languages". Symposium on Static Analysis. Springer, 1999.

[6] A. Bouakaz and J.-P. Talpin. "Buffer minimization in earliest-deadline first scheduling of dataflow graphs". Conference on Languages, Compilers and Tools for Embedded Systems. ACM, 2013.

[7] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. "Theory of Latency-Insensitive Design". Transactions on Computer-Aided Design of Integrated Circuits and Systems, v. 20(9). IEEE, 2001.

[8] J.-L. Colaco, A. Girault, G. Hamon, and M. Pouzet. "Towards a Higher-order Synchronous Data-flow Language". Int. Conf. on Embedded Software. ACM, 2004.

[9] P. Feautrier, A. Gamatie and L. Gonnord. "Enhancing the compilation of synchronous data-flow programs with a combined numerical-boolean abstraction'. Journal of Computing. Computer Society of India, 2012.

[10] J. Forget, F. Boniol, D. Lesens and C. Pagetti. A Real-Time Architecture Design Language for Multi-Rate Embedded Control Systems. Symposium on Applied Computing. ACM, 2010.

[11] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud. "The synchronous data flow programming language LUSTRE," Proc. of the IEEE, v.79, 1991.

[12] G. Kahn. "The semantics of a simple language for parallel programming". Proceedings of the IFIP Congress on Information Processing. IFIP, 1974.

[13] E.A. Lee, D.G. Messerschmitt. "Synchronous data flow." Proceedings of the IEEE, v.75, 1987.

[14] Le Guernic, P., Talpin, J.-P., Le Lann, J.-C. "Polychrony for system design". Journal for Circuits, Systems and Computers. World Scientific, 2003.

[15] R. von Hanxleden, et al. "SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications". Conference on Programming Language Design and Implementation. ACM, 2014.

[16] J. Kloos, R. Majumdar, V. Vafeiadis. "Asynchronous Liquid Separation Types". European Conf. on Object-Oriented Programming. LIPICS, 2015.

[17] O. Maffeis, P. Le Guernic. "Distributed Implementation of Signal: Scheduling and Graph Clustering". Formal Techniques in Real-Time and Fault-Tolerant Systems, LNCS v. 863. Springer, 1994.

[18] L. Mandel, F. Plateau, and M. Pouzet. "Static Scheduling of Latency Insensitive Designs with Lucy-n". In International Conference on Formal Methods in Computer-Aided Design, 2011.

[19] S. Merz, H. Vanzetto. "Refinement Types for TLA+". NASA Symposium on Formal Methods. Springer, 2014.

[20] L. de Moura and N. Bjorner. "Z3: An efficient SMT solver". International conference on Tools and algorithms for the construction and analysis of systems. Springer, 2008. https://z3.codeplex.com

[21] C. Ngo, J.-P. Talpin, T. Gautier. "Efficient deadlock detection for polychronous data-flow specifications". Electronic System Level Synthesis Conference. IEEE, 2014.

[22] D. Potop-Butucaru, Y. Sorel, R. de Simone, JP. Talpin. "From concurrent multi-clock programs to deterministic asynchronous implementations". Fundamenta Informaticae. IOS Press, 2011.

[23] M. Pouzet, P. Raymond. "Modular Static Scheduling of Synchronous Data-flow Networks – An efficient symbolic representation". International Conference on Embedded Software. ACM, 2009.

[24] A. Pnueli, M. Siegel, E. Singerman. "Translation validation". Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 1998.

[25] P. M. Rondon, M. Kawaguchi, R. Jhala. "Liquid Types". Conference on Programming language Design and Implementation. ACM, 2008.

[26] I. Smarandache, T. Gautier and P. Le Guernic. "Validation of mixed Signal-Alpha real-time systems through an affine calculus on clock synchronisation constraints". World Congress on Formal Methods in the Development of Computing Systems. Springer, 1999.

[27] Talpin, J.-P., Nowak, D. "A synchronous semantics of higher-order processes for modeling reconfigurable reactive systems". Foundations of Software Technology and Theoretical Computer Science. Springer, 1998.

[28] J.-P. Talpin, J. Brandt, M. Gemünde, K. Schneider, and S. Shukla. "Constructive Polychronous Systems". In Science of Computer Programming. Elsevier, 2014.

[29] J.-P. Talpin, P. Jouvelot, S. Shukla. "Liquid clocks: refinement types for time-dependent stream functions". Technical Report n. 8747. INRIA, June 2015. Available from https://hal.archives-ouvertes.fr/hal-01166350v3.

[30] The Liquid Haskell project. "LiquidHaskell, Refinement Types via SMT and Predicate Abstraction". http://goto.ucsd.edu/~rjhala/liquid.

[31] N. Vazou, P. M. Rondon, R. Jhala. "Abstract refinement types". European conference on Programming Languages and Systems. Springer, 2013.

[32] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, S. Peyton-Jones. "Refinement Types For Haskell". SIGPLAN Int. Conf. on Functional Programming. ACM, 2014.