Clock Analysis of Synchronous Dataflow Programs

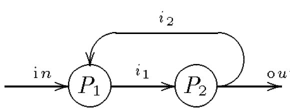
Thomas P. Jensen
CNRS-LIX
Ecole Polytechnique
91128 Palaiseau
France
Email: jensen@lix.polytechnique.fr

Abstract

Synchronous dataflow languages such as Lustre and Signal have been proposed as a tool for programming reactive systems. These languages rely on a clock analysis to ensure that synchronous operations receive their arguments at the same time. We present a denotational model of a Lustre-like dataflow language and show how a range of clock analyses for this language can be designed and proved correct. To the best of our knowledge this is the first formal correctness proof for such an analysis. We then give a type system formulation of this analysis using clocks as types and show how adding polymorphic clocks enables us to treat programs where clocks vary according to their context. The relationship to the clock analysis by constraint solving used in the language Signal is discussed.

1 Introduction

One of the earliest models of concurrent computation is Kahn's networks of processes [Kah74, KM77]. Such a network consists of a set of deterministic processes, or agents, communicating in an asynchronous fashion via unbounded channels. Input consists of streams of data that arrives on designated input channels. Each data token is sent through the network resulting in a stream of data appearing on the output channels. The assumption of asynchronous communication and unbounded channels means that the relative speed at which the agents proceeds can be ignored in the model—if one agent produces data quicker than another agent can consume it then this data is just stored in the unbounded channels. These simplifying assumptions are mirrored in the ease with which the semantics of a network can be defined as a fixed point of a set of recursive dataflow equations induced in a straightforward manner by the network. For example, the network



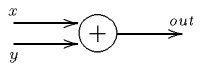
To appear in Proc. of ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation, June 1995, San Diego CA.

gives rise to the following set of equations

$$\begin{cases} i_1 &= P_1(in, i_2) \\ i_2 &= P_2(i_1) \\ out &= P_2(i_1) \end{cases}$$

whose solution can be determined as a fixed point over the complete partial order of streams of data elements. We call this solution the dataflow semantics of the network.

It would be desirable to adapt this model to the realm of reactive real-time systems which usually contain some degree of parallel activity. Viewed as a reactive system, a network would receive input data at time n on its input channels, process this data and emit the result on the output channels before input at time $n+1$ arrive. If some input to a computation is not available at that moment, the result of that computation is undefined; the system doesn't wait for data to arrive. The following simple example demonstrates the difference between the two views of the same network:



In the untimed dataflow view, the result of sending the streams $x = x_1x_2x_3\ldots$ and $y = y_1y_2y_3\ldots$ yields the output stream $x_1+y_1, x_2+y_2, x_3+y_3, \ldots$ In the reactive view, this is only the case if x_i and y_i arrive at the same time *i.e.*, if the two input streams are synchronous. If x is twice as fast as y, the output will be $x_2+y_1, x_4+y_2, x_6+y_3, \ldots, x_{2n}+y_n$ which would arrive with the same speed as y.

The question is now: what restrictions can we impose on a network and its input flows so that the dataflow semantics and the reactive semantics coincide? One solution is presented in the Lustre synchronous programming language [CPHP87]. A Lustre program describes a network where the assumptions are that the n-th input tokens arrive at the same moment in time at all agents and that the computations and communications taking place at and in between the agents are instantaneous, compared to the time between the arrivals of two sets of input. These assumptions of *synchrony* implies that dataflow semantics and reactive semantics coincide. See [BB91] and [Hal93] for general introductions to synchronous programming.

In order to verify that the flows of arguments to synchronous operations have the same clock, Lustre uses a *clock analysis* to calculate the clock of every flow in a program. Lustre offers operations for manipulating the clocks of flows which makes the clock calculus undecidable and an approximate clock analysis is therefore called for. Two such analyses have been proposed by Bergerand [Ber86] and Plaice [Pla88]. The present paper presents a framework for designing such analyses and for proving them correct.

We stress that the goal of clock analysis as it is employed in current languages is to check the validity of a program rather than to do program optimisations as is usually the case with program analyses. The situation can be compared to type checking of statically typed languages where an analysis is used to accept or reject programs at compile-time.

This paper presents a language $\mathcal{L}$ for manipulating timed dataflows (Section 2) and its denotational semantics (Section 3). This defines rigorously the notion of timed dataflow presented above. The semantics assigns a dataflow and a clock to every expression in an $\mathcal{L}$ program. An abstract interpretation [CC77] is used to extract an approximation to the clocks of the flows of a program (Section 6) and we prove that the information calculated by the analysis is always a safe approximation of the semantics of the program. In Section 7 we present a type system formulation of the clock analysis and discuss how this analysis relates to an earlier analysis proposed by Bergerand [Ber86]. By extending the type system with type variables that represent clocks that can vary according to their context we obtain a polymorphic clock analysis (Section 8) which we use to relate the clock analysis in the SIGNAL language [LB86, LGLL91] to the analyses discussed here.

2 A Synchronous Dataflow Language

In this section we define a synchronous dataflow language $\mathcal{L}$ and provide examples of programs written in $\mathcal{L}$. [1] Assume given a set of variables $x_1, x_2, \ldots$. The set of expressions of $\mathcal{L}$ is given by the grammar

$$\begin{array}{rcl} e & = & k \mid x \mid e_1 \; op \; e_n \mid \\ & & \mathsf{pre}(e) \mid e_1 \rightarrow e_2 \mid \\ & & e_1 \;\mathsf{when}\; e_2 \mid \mathsf{current}(e) \end{array}$$

where k ranges over integer and boolean constants and f ranges over integer and boolean operations such as

$$+, -, =, \wedge, \mathsf{mod}\; etc.$$

The operator pre is the delay operator. It memorises the last value of a flow and outputs it when it receives a new value, transforming a sequence $d_1 d_2 \ldots$ with clock κ into the sequence $* d_1 d_2 \ldots$ with the same clock. Here $*$ is a special symbol that denotes the absence of a value.[2] The initialisation operator $\rightarrow$ maps flows $(d_1 d_2 \ldots, \kappa)$ and $(c_1 c_2 \ldots, \kappa)$ to the flow $(d_1 c_2 c_3 \ldots, \kappa)$. The $\rightarrow$ operator only gives well-defined output as long as the input have the same clock.

The expression e when b samples values from e when b is true. Here e and b must be on the same clock and b must be a boolean flow. The clock of the flow defined by e when b consists of those instants where b is true. An example in schematic form:

$$\begin{array}{rcllllllll} e & = & e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 \\ b & = & f & t & t & f & t & f & f & t \\ e \;\mathsf{when}\; b & = & & e_2 & e_3 & & e_5 & & & e_8. \end{array}$$

The operator current performs up-sampling, or interpolation, of a flow. The clock of $\mathsf{current}(e)$ is *the clock of the clock of* e. This clock will always be faster or equal to the clock of e. At the instants where e is available, $\mathsf{current}(e)$ takes the value of e. When e is not available, $\mathsf{current}(e)$ takes the last available value of e. Continuing the example from above, $\mathsf{current}(e \;\mathsf{when}\; b))$ will then be the flow

$$\mathsf{current}(e \;\mathsf{when}\; b)) \;=\; * \quad e_2 \quad e_3 \quad e_3 \quad e_5 \quad e_5 \quad e_5 \quad e_8.$$

A *node* consists of a declaration of input variables and a set of equations defining the output flows. It has the format

$$\begin{array}{rcl} \multicolumn{3}{l}{\mathsf{node}\ (y_1, \ldots, y_k)} \\ x_1 & = & e_1 \\ & \vdots & \\ x_n & = & e_n \end{array}$$

The following node is the standard example of how to define a counter in $\mathcal{L}$:

$$\begin{array}{rcl} \multicolumn{3}{l}{\mathsf{node}\ ()} \\ C & = & 0 \rightarrow 1 + \mathsf{pre}(C) \end{array}$$

For a more elaborate example we consider the programming of a monitoring device that each second receives a numerical value through input *In*. Every minute the device memorises the value received and it raises an alarm if the values received during the following minute deviate too much from the memorised value. Assuming that we have a flow *Sec* that gives the number of seconds elapsed since the beginning this can be programmed as follows:

$$\begin{array}{lcl} \multicolumn{3}{l}{\mathsf{node}(Sec : int, In : int)} \\ NewMin & = & (Sec \;\mathsf{mod}\; 60 = 0) \\ InS & = & In \;\mathsf{when}\; NewMin \\ Alarm & = & |\mathsf{current}(InS) - In| > Critical \end{array}$$

The flow *InS* is obtained by sampling *In* every minute. The up-sampling $\mathsf{current}(InS)$ gives access to the memorised value at every second since its clock is the clock of the clock of *InS* *i.e.*, the clock of *NewMin* which is the clock of *In* and *Sec*. The two flows *In* and $\mathsf{current}(InS)$ are therefore synchronised. The flow *Alarm* becomes *true* when the difference between the values of the two flows is greater than a given value *Critical*.

3 Denotational semantics

The denotational semantics of $\mathcal{L}$ is based on the semantics for LUSTRE defined by Plaice [Pla88] which in turn is an extension of Kahn's original semantics for dataflow networks [Kah74]. The difference between our semantics and that of Plaice is in the definition of the sampling operators and the presentation of the underlying domains. Bergerand's

[1] The syntax of $\mathcal{L}$ is taken from the language LUSTRE but we warn the reader that not all $\mathcal{L}$ programs considered here will be accepted as correctly timed LUSTRE programs.

[2] Other presentations use $\perp$ to denote absence of data. This can however be misleading since absence is treated as a value just like any other value—in particular it is not smaller than other values in the domain ordering.

semantics [Ber86] is only defined for programs that have passed a clock analysis and does not deal with timed flows.

The basic data domains consists of finite and infinite sequences over the sets of integer and boolean values extended with the value $*$ to represent the absence of a value. Sequence s_1 approximates s_2 if s_1 is a prefix of s_2, written $s_1 \leq s_2$. Thus our domains are

$$(D \cup \{*\})^\omega = (D \cup \{*\})^* \cup (D \cup \{*\})^\infty$$

where D ranges over *Bool* and *Int*, D^* is the set of finite sequences over D and D^∞ the set of infinite sequences over D.

3.1 Clocks and flows.

When modelling clocks we assume that there is a common clock that defines the basic clock frequency and ticks forever. An arbitrary clock is specified by a sequence of ticks and non-ticks understood to be relative to the common clock, *e.g.*, the clock that ticks every third instant is modelled by the sequence

$$\top\bot\bot\top\bot\bot\ldots.$$

The corresponding domain is thus

$$2^\omega = \{\bot, \top\}^\omega$$

with the common clock modelled by the infinite sequence of ticks:$\top^\infty$. We admit finite sequences as clocks of flows that only have a finite duration. The ordering on clocks is the prefix ordering, $\leq$, reflecting the fact that more information about a clock is provided by defining it for more instants. In particular, the undefined clock is the empty sequence, denoted by ϵ, which is least in the $\leq$ ordering.

In the more general case, clocks themselves can have clocks if they have been defined via sampling. In order to describe the operators when and current it is necessary to keep track of from which clocks a clock has been defined. We follow the approach of [Pla88] and define our general clock domain to be lists, possibly infinite, of sequences over $\{\bot, \top\}$, *i.e.*,

$$\mathit{Clock} = (\mathit{List}(2^\omega), \sqsubseteq)$$

where $\sqsubseteq$ denotes the standard ordering on lists *i.e.*, $l \sqsubseteq m$ if for all elements l_i there is a corresponding element m_i in m such that l_i is smaller than m_i. We call elements $\kappa \in \mathit{Clock}$ general clocks or just clocks and reserve the term *simple clocks* for the elements $\tau \in 2^\omega$. The *Clock* element $[\tau_1, \ldots, \tau_n]$ represents a clock that ticks as defined by the simple clock τ_1 and has been sampled using the clocks $\tau_2, \ldots, \tau_n$.

Flow operators such as $+$ and and only produce output as long as their input arguments arrive at the same time. The longest period of time for which two simple clocks, τ_1 and τ_2, stay synchronous corresponds to their longest common prefix which is exactly their greatest lower bound, $\tau_1 \sqcap \tau_2$ in the prefix ordering. Generalising this to general clocks, κ_1 and κ_2, we define their synchronisation to be the clock obtained by applying $\sqcap$ pointwise to κ_1 and κ_2:

$$\begin{aligned} \kappa_1 \sqcap \kappa_2 &\equiv [\tau_{11}, \tau_{12}, \ldots] \sqcap [\tau_{21}, \tau_{22}, \ldots] \\ &= [\tau_{11} \sqcap \tau_{21}, \tau_{12} \sqcap \tau_{22}, \ldots, \tau_{1n} \sqcap \tau_{2n} \ldots]. \end{aligned}$$

such that the length of $\kappa_1 \sqcap \kappa_2$ is equal to the length of the shortest of the lists κ_1 and κ_2. This means that for two clocks $[\tau_{11}, \tau_{12}, \ldots]$ and $[\tau_{21}, \tau_{22}, \ldots]$ to be synchronous we require not only that τ_{11} and τ_{21} are synchronous but also that the clocks they have been sampled from are synchronous. For example, the synchronisation of the clocks

$$[\bot\bot\bot\top, \top\top\top\top] \quad \text{and} \quad [\bot\bot\bot\top, \bot\top\bot\top, \top\top\top\top]$$

yields the clock

$$[\bot\bot\bot\top, \epsilon]$$

representing the fact that it is not possible to find a common clock for the sampling of the two clocks.

A flow of values from a set D is then a pair consisting of a sequence over D and a clock:

$$\mathit{Flow}^D = (D \cup \{*\})^\omega \times \mathit{Clock}.$$

An element $(d, [\tau_1, \tau_2, \ldots])$ represents the flow that produces the i-th element of d at the instant when the i-th tick of τ_1 appears. The operation $|\ ,\ |$ yields the sequence obtained by looking at the flow $(d, \kappa) = (d, [\tau_1, \ldots, \tau_n])$ at all instants. The sequence obtained is d "padded out" with $*$ values when τ_1 doesn't tick. We first define it for simple clocks by

$$|d, \tau| = \begin{cases} hd(d) :: |tl(d), tl(\tau)| & \text{if } hd(\tau) = \top \\ * :: |d, tl(\tau)| & \text{if } hd(\tau) = \bot \\ \epsilon & \text{if } \tau = \epsilon \end{cases}$$

and extend it to general clocks by stipulating that $|d, \kappa| = |d, hd(\kappa)|$. For the flow $(1^\infty, [(\bot\bot\top)^\infty])$ that produces a 1 every third instant we have

$$|1^\infty, (\bot\bot\top)^\infty| = * * 1 * * 1 * * 1\ \ldots$$

We shall also need to change the clock of a flow. Regarding an infinite sequence of data d as a flow available at all instants (clock $= \top^\infty$), $d \downarrow \tau$, yields the sequence of d_i's that are observed by only looking at d when the simple clock $\tau \in 2^\omega$ ticks. With this operation we can re-time any flow by first converting it to a flow on $\top^\infty$ using $|\_, \_|$ and then restrict it to the new clock using $\downarrow$. Formally, $\downarrow$ is defined by:

$$d \downarrow \tau = \begin{cases} hd(d) :: tl(d) \downarrow tl(\tau) & \text{if } hd(\tau) = \top \\ tl(d) \downarrow tl(\tau) & \text{if } hd(\tau) = \bot \\ \epsilon & \text{if } \tau = \epsilon \end{cases}$$

As with $|\_, \_|$ we extend $\downarrow$ to general clocks by defining

$$d \downarrow \kappa = d \downarrow hd(\kappa).$$

To extract a simple clock from a boolean flow we introduce the operation *True* that converts an infinite boolean sequence into a clock that ticks exactly when that boolean sequence has the value *true*. $\mathit{True}(d)$ is the sequence obtained by applying the function $\lambda d_i.\mathit{if}\ d_i = \mathit{true}\ \mathit{then}\ \top\ \mathit{else}\ \bot$ to each element of the sequence d. It will not be defined further. Finally, for up-sampling we define the operator

$$\Sigma : (D \cup \{*\})^\infty \to (D \cup \{*\})^\infty$$

that maps the infinite sequence d into the sequence defined by

$$(\Sigma(d))_i = \begin{cases} d_i & \text{if } d_i \neq * \text{ or } i = 0 \\ (\Sigma(d))_{i-1} & \text{otherwise.} \end{cases}$$

The sequence $\Sigma(d)$ is obtained by replacing every $*$ in d by the last non-$*$ element occurring before the $*$ in d, if such an element exists. For example, by applying Σ to the sequence

$$d = * * d_1 * * d_2 * * d_3 \ \ldots$$

we would obtain

$$\Sigma(d) = * * d_1\ d_1\ d_1\ d_2\ d_2\ d_2\ d_3\ \ldots.$$

3.2 Semantics of expressions

The semantics of $\mathcal{L}$ expressions is defined by the function

$$[\![\ ]\!] : Exp \rightarrow Env \rightarrow Flow$$

where Env is an environment mapping variables to flows. We now go through the semantic definition of each of the $\mathcal{L}$ constructs letting (d_i, κ_i) stand for the value $[\![e_i]\!]\rho$.

A constant k denotes the flow that consists of the sequence $(k, k, k, \ldots)$, written k^∞. Its clock is the common clock $[\top^\infty]$.

$$[\![k]\!]\rho = (k^\infty, [\top^\infty])$$

Variables, both input and local, are looked up in the environment ρ:

$$[\![x]\!]\rho = \rho(x).$$

The predefined arithmetic and logical operators are all synchronous *i.e.*, they require that all arguments have the same clocks which is then the clock of the result. The resulting data sequence is obtained by applying the operator op pointwise to the sequences, so that the i-th element is $d_{1i}\ op\ d_{2i}$ if all sequences contain at least i elements, and $*$ otherwise. We overload op and write this sequence as $d_1\ op\ d_2$.

$$[\![e_1\ op\ e_2]\!]\rho = (d_1\ op\ d_n), \kappa_1 \sqcap \kappa_2).$$

The delay operator pre delays a flow by prefixing its data sequence with a $*$. The clock is the same as that of the argument.

$$[\![\mathsf{pre}(e)]\!]\rho = (* :: d, \kappa).$$

The initialisation operator $\rightarrow$ takes first element of e_1's sequence and then continues as e_2. Following most existing languages the clocks of e_1 and e_2 are required to synchronise.

$$[\![e_1 \rightarrow e_2]\!]\rho = (hd(d_1) :: tl(d_2), \kappa_1 \sqcap \kappa_2).$$

where hd and tl are the usual list operations that returns the first element and removes the first element respectively. We define that tl of the empty list is the empty list. It can be argued that it would be reasonable only to require that e_1 and e_2 synchronise on the first element since e_1 is then ignored[3]. The modifications needed to accommodate this point of view are straightforward.

In order to define the semantics of e_1 when e_2 we assume that the semantics of the subexpressions e_1, e_2 is given as $[\![e_i]\!]\rho = (d_i, [\tau_{i1}, \tau_{i2}, \ldots])$ for $i = 1, 2$. The clocks of e_1 and e_2 are still required to be synchronous. The new clock is $\tau = True(\mid d_2, \kappa_{21} \mid)$.

$$[\![e_1\ \mathsf{when}\ e_2]\!]\rho = (|d_1, \tau_{11}| \downarrow \tau, \tau :: \kappa_1 \sqcap \kappa_2)$$

With $(d, [\tau_1, \tau_2, \ldots]) = [\![e]\!]\rho$ the semantics of current(e) is

$$[\![\mathsf{current}(e)]\!]\rho = (\Sigma(\mid d, \tau_1 \mid) \downarrow \tau_2, [\tau_2, \tau_3 \ldots]).$$

Notice how the clock of the clock of e is obtained by taking the tail of the clock of e.

[3]This would correspond to the **$1-init** operator from SIGNAL.

3.3 Semantics of nodes

We specify the semantics of a node

$$\mathsf{node}\ N(x_1, \ldots x_k)\{x_j = e_j\}_{j=k+1,\ldots,n}$$

as the solution of the set of recursive equations $\{x_j = e_j\}$ given the flows associated to the input variables $x_1, \ldots, x_k$. The solution can be expressed as the least fixed point of a function on flows defined by the expressions $e_{k+1}, \ldots, e_n$. This function $F : Flow^n \rightarrow Flow^n$ is defined by

$$F(f_1, \ldots, f_n) = (f_1, \ldots, f_k, [\![e_{k+1}]\!]\rho, \ldots, [\![e_n]\!]\rho).$$

where $\rho(x_j) = f_j$ for $j = 1, \ldots, n$. Given the input flows the fixed point can then be calculated in the standard way as the least upper bound of a sequence of iterations:

$$[\![\mathsf{node}\ N(x_1, \ldots x_k)\{x_j = e_j\}]\!] = \lambda f_1, \ldots, f_k. \bigsqcup_{i=0}^{\infty} F^i(f_1, \ldots, f_k, (\epsilon, [\top^\infty]), \ldots, (\epsilon, [\top^\infty])).$$

This semantics yields as results the tuple of all the x_j-flows. In existing languages like LUSTRE it is possible to specify that a certain number of the x_j's are local to the node and not available outside the node. This form of hiding can be modelled by projecting out the local flows from the semantics of the node. We shall not be concerned with this aspect of the language for the remainder of the paper and therefore we leave it out of the semantic definition.

4 Abstract domains of clocks

In the following sections we consider the problem of finding an approximate characterisation of the clock of a given expression e by abstract interpretation of e. Our first concern is to find lattices of clock properties that provide approximate descriptions of clocks. We show how lattices of general clocks and tuples of general clocks can be constructed in a systematic manner from an initial lattice of properties of simple clocks.

4.1 Abstract clocks

We shall describe the relationship between concrete and abstract clocks through a relation $\sim \subseteq 2^\omega \times \widehat{2^\omega}$ where $\widehat{2^\omega}$ is a complete lattice of clock properties that we want to analyse for. The idea is that $\kappa \sim \widehat{\kappa}$ if $\widehat{\kappa}$ is a property of κ. Some constraints on the lattice $\widehat{2^\omega}$ and relation $\sim$ are necessary for the construction of the theory; they are given in the following definition:

Definition 4.1 (Abstraction of simple clocks) *An* abstraction *of simple clocks is a complete lattice $\widehat{2^\omega}$ and a relation $\sim \subseteq 2^\omega \times \widehat{2^\omega}$ such that*

1. $\forall \tau \in 2^\omega : \tau \sim \top$
2. *Let $\{\tau^i\}_{i \in \mathcal{N}} \subseteq 2^\omega$ and $\{\widehat{\tau}^i\}_{i \in \mathcal{N}} \subseteq \widehat{2^\omega}$ be chains. If $\tau^i \sim \widehat{\tau}^i$ for all $i \in \mathcal{N}$ then*

$$\bigsqcup_{i \in \mathcal{N}} \tau^i \sim \bigsqcup_{i \in \mathcal{N}} \widehat{\tau}^i.$$

$$
\begin{array}{lcll}
[\![k]\!]\rho & = & (k^\infty, [\top^\infty]) & \\
[\![x]\!]\rho & = & \rho(x) & \\
[\![e_1\ op\ e_2]\!]\rho & = & (d_1\ op\ d_2, \kappa_1 \sqcap \kappa_2) & \\
[\![\mathsf{pre}(e)]\!]\rho & = & (* :: d, \kappa) & \\
[\![e_1 \rightarrow e_2]\!]\rho & = & (hd(d_1) :: tl(d_2), \kappa_1 \sqcap \kappa_2) & \\
[\![e_1\ \mathsf{when}\ e_2]\!]\rho & = & (|d_1, \tau_{11}| \downarrow \tau, \tau :: \kappa_1 \sqcap \kappa_2) & \text{where } \tau = True(|\ d_2, \kappa_{21}\ |) \\
[\![\mathsf{current}(e)]\!]\rho & = & (\Sigma(|\ d, \tau_1\ |) \downarrow \tau_2, tl(\kappa)) & \text{where } \tau_1 = hd(\kappa), \tau_2 = hd(tl(\kappa))
\end{array}
$$

$$
\begin{array}{l}
[\![\mathsf{node}\ N(x_1, \ldots x_k)\{x_j = e_j\}]\!] = \lambda f_1, \ldots, f_k. \bigsqcup_{i=0}^{\infty} F^i(f_1, \ldots, f_k, (\epsilon, [\top^\infty]), \ldots, (\epsilon, [\top^\infty])) \\
\quad \text{where } F(f_1, \ldots, f_n) = (f_1, \ldots, f_k, [\![e_{k+1}]\!][x_j \mapsto f_j], \ldots, [\![e_n]\!][x_j \mapsto f_j])
\end{array}
$$

Figure 1: The $\mathcal{L}$ semantics

Condition *1.* guarantees that there is an element for representing that we have no information about a clock. A relation on cpos satisfying *2.* is said to be *inclusive.* As examples of abstractions of simple clocks we have the following lattices:

Example 4.2

$$
\widehat{(2^\omega)} = \omega^\top : \qquad \begin{array}{c} \top \\ | \\ \infty \end{array}
$$

with concretisation relation

$$
\top^\infty \sim \infty \qquad \textit{and} \qquad \forall \tau \in 2^\omega : \tau \sim \top.
$$

With this abstraction we can only represent that a flow is present at all instants; all other information is lost (approximated by the empty property $\top$). In particular all finite clocks are approximated by $\top$. A more refined analysis could represent the fact that a clock ticks during the first n instants. It would use as abstraction of simple clocks the lattice of ordinals up to the first infinite ordinal, denoted by ∞, augmented with a top element.

Example 4.3

$$
\widehat{(2^\omega)} = \mathbf{Ord}^\top : \qquad \begin{array}{c} \top \\ | \\ \infty \\ \vdots \\ n+1 \\ | \\ n \\ \vdots \\ 0 \end{array}
$$

The concretisation relation is given by

$$
\top^n \sim n \qquad \top^\infty \sim \infty \qquad \textit{and} \qquad \forall \tau \in 2^\omega : \tau \sim \top.
$$

The element n represents the clock that ticks every instant for the first n instants. As in the previous example ∞ concretises to the clock $\top^\infty$.

4.2 Symbolic clocks

Sampling turns a boolean flow into a clock. Since we haven't put data information into our analysis we would have to abstract the result of a sampling by the "no information" element $\top$. Any later synchronisation with such a clock would therefore yield $\top$. As observed already in [Ber86] we can do slightly better by noticing that when synchronising two clocks we do not need to know their respective value if we know by some other means that those values are in fact the same. This is for instance the case when the clocks are defined by the same variable as in the expressions e_1 when x and e_2 when x. The more general case where two clocks are defined by the same expression in the same environment can be reduced to the simpler case by introducing a new local definition *i.e.*, by the transformation

$$
\left\{ \begin{array}{l} x_1 = e_1\ \mathsf{when}\ b \\ x_2 = e_2\ \mathsf{when}\ b \end{array} \right. \quad \Rightarrow \quad \left\{ \begin{array}{l} x_1 = e_1\ \mathsf{when}\ x_3 \\ x_2 = e_2\ \mathsf{when}\ x_3 \\ x_3 = b. \end{array} \right.
$$

This transformation is standard and is detailed in [Pla88]. The formalisation of the above ideas requires the following construction.

Definition 4.4 *Let $(L, \sqsubseteq_L)$ be a lattice and X a set of variables. Define the lattice*

$$
L[X] = (L \cup X, \sqsubseteq)
$$

where

$$
a \sqsubseteq b \Leftrightarrow \left\{ \begin{array}{l} a, b \in L \textit{ and } a \sqsubseteq_L b \\ a = \bot_L \textit{ and } b \in X \\ a = b \\ a \in X \textit{ and } b = \top_L \end{array} \right.
$$

The variables in X are thus incomparable with the original elements of L, except for $\bot_L$ and $\top_L$. Their intended use is to record that some property of an expression varies as a function of the value of a variable. This enables us to represent that two expressions will satisfy the same property in every environment by saying that they satisfy the same variable-dependent property. Notice that this doesn't say what the property is. For our purpose we shall use this to represent that two expressions have the same clock even though we don't know what this clock is.

We extend the abstraction relation to the abstract domain $\widehat{2^\omega}[X]$ by parameterising the abstraction relation on a tuple $f = (f_1, \ldots, f_n)$ of flows with the intention that f_i is the flow associated with x_i. When used to represent a property of clocks the variables x_i stand for the simple clock defined by the flow of x_i, *i.e.*, the simple clock $True(|f_i|)$. We can then define the concretisation relation $\sim_f$ on simple clocks as follows:

Definition 4.5 (Abstraction of symbolic clocks) *Let* $\sim \subseteq 2^\omega \times \widehat{2^\omega}$ *be an abstraction of simple clocks and let* $f = (f_1, \ldots, f_n) \in \mathrm{Flow}^n$. *Define the relation* $\sim_f \subseteq 2^\omega \times \widehat{2^\omega}[X]$ *as follows:*

- $\tau \sim_f \widehat{\tau}$ *if* $\widehat{\tau} \in \widehat{2^\omega}$ *and* $\tau \sim \widehat{\tau}$
- $True(|f_i|) \sim_f x_i$

4.3 Abstraction of general clocks

General clocks are lists of simple clock. There are numerous ways of abstracting lists in order to deal with the infinite list domain but we shall not commit our developments to a particular approach at this stage. Instead we take the general abstract domain of clocks to be lists over the abstract domain of simple clocks. The abstraction relation on lists is taken to be the elementwise extension of the relation on simple clocks. We overload the symbol $\sim_f$ to denote the abstraction relation on both simple and general clocks.

Definition 4.6 (Abstraction of general clocks) *Let* $\sim_f \subseteq 2^\omega \times \widehat{2^\omega}[X]$ *be an abstraction of simple clocks. Define the abstract domain of general clocks by*

$$\widehat{Clock} = List(\widehat{2^\omega})$$

and the concretisation relation $\sim_f \subseteq Clock \times \widehat{Clock}$ *by:*

$$\kappa \sim_f \widehat{\kappa} \Leftrightarrow \begin{cases} length(\kappa) = length(\widehat{\kappa}) \\ \forall\, i : \kappa_i \sim_f \widehat{\kappa}_i. \end{cases}$$

The next lemma states that $\kappa \sim_f \widehat{\kappa}$, viewed as a ternary predicate, is inclusive.

Lemma 4.7 *Let* $\{\kappa^i\}_{i\in\mathcal{N}} \subseteq Clock, \{\widehat{\kappa}^i\}_{i\in\mathcal{N}} \subseteq \widehat{Clock}$ *and* $\{f^i\}_{i\in\mathcal{N}} \subseteq Flow^n$ *be chains with limits* $\kappa, \widehat{\kappa}$ *and* f *respectively. Assume that* $\kappa^i \sim_{f^i} \widehat{\kappa}^i$ *for all* $i \in \mathcal{N}$. *Then* $\kappa \sim_f \widehat{\kappa}$.

Proof From the assumption

$$\forall i \in \mathcal{N} : \kappa^i \sim_{f^i} \widehat{\kappa}^i \tag{1}$$

it follows that

$$length(\kappa^i) = length(\widehat{\kappa}^i)$$

and therefore since *length* is continuous,

$$length(\kappa) = length(\widehat{\kappa}).$$

It remains to prove that κ and $\widehat{\kappa}$ are related elementwise. Let τ and $\widehat{\tau}$ be the k-th element of κ and $\widehat{\kappa}$ respectively. Then there exists $N \in \mathcal{N}$ such that

$$\tau = \bigsqcup_{i>N} \tau^i \quad \text{and} \quad \widehat{\tau} = \bigsqcup_{i>N} \widehat{\tau}^i$$

where τ^i and $\widehat{\tau}^i$ are the k-th element of the i-th approximant. From the hypothesis (1) it follows that

$$\tau^i \sim_{f^i} \widehat{\tau}^i. \tag{2}$$

By case analysis on $\widehat{\tau}$ we now show that $\tau \sim_f \widehat{\tau}$.

$\widehat{\tau} = \top$. Trivially.

$\widehat{\tau} = x_k \in X$. Then from some point on in the chain of abstract clocks we have that $\widehat{\tau}^i = \widehat{\tau}^{i+1} = \ldots = x_k$ which, by (2), implies that $\tau^i = True(|f_k^i|), \tau^{i+1} = True(|f_k^{i+1}|), \ldots$ and therefore, by continuity of the involved operators, that

$$\tau = \bigsqcup_{i=0}^{\infty} \tau^i = \bigsqcup_{i=0}^{\infty} True(|f_k^i|) = True(|f_k|) \sim_f x_k.$$

$\widehat{\tau} \in \widehat{2^\omega}$. Then every $\widehat{\tau}^i$ belongs to $\widehat{2^\omega}$ and from Definition 4.5 we have that $\tau^i \sim \widehat{\tau}^i$. Since $\sim$ is assumed inclusive we get that $\tau \sim \widehat{\tau}$.

□

4.4 Tuples of clocks

In order to state the correctness of the abstract interpretation of nodes we define a relation γ that states when a tuple of flows is correctly described by a tuple of abstract clocks. Due to the presence of symbolic clocks in the abstract domains we can have abstract clocks of the form

$$(\ldots, [x_j], \ldots, [x_j], \ldots)$$

which describe those tuples of flows where the clocks of the corresponding components are the same and determined by the flows of the j-th component. This consistency condition on a tuple f of flows is expressed by using f itself to define the concretisation of symbolic clocks:

Definition 4.8 (Abstraction relation for tuples) *Let* $f = (f_1, \ldots, f_n) \in Flow^n$ *and* $\kappa = (\kappa_1, \ldots, \kappa_n) \in \widehat{Clock}^n$. *Define the relation* $\gamma \subseteq Flow^n \times \widehat{Clock}^n$ *by*

$$f \;\gamma\; \kappa \;\equiv\; \chi(f_i) \sim_f \kappa_i \qquad \text{for } i = 1, \ldots, n$$

Notice that the interdependencies between the elements in the tuple is expressed solely through using the same f when concretising the different elements. If no $a_i \in X$ then this corresponds to the ordinary use of cartesian product in abstract interpretation to represent non-relational properties of tuples.

$$
\begin{array}{lcl}
\widehat{[\![k]\!]}\rho & = & [\omega] \\
\widehat{[\![x]\!]}\rho & = & \rho(x) \\
[\![\widehat{e_1\ op\ e_2)}]\!]\rho & = & \widehat{[\![e_1]\!]}\rho \ \S\ \widehat{[\![e_2]\!]}\rho \\
[\![\widehat{\mathsf{pre}(e)}]\!]\rho & = & \widehat{[\![e]\!]}\rho \\
[\![\widehat{e_1 \rightarrow e_2}]\!]\rho & = & \widehat{[\![e_1]\!]}\rho \ \S\ \widehat{[\![e_2]\!]}\rho \\
[\![e_1 \ \widehat{\mathsf{when}}\ x]\!]\rho & = & x :: (\widehat{[\![e_1]\!]}\rho \ \S\ \rho(x)) \\
[\![\widehat{\mathsf{current}}(e)]\!]\rho & = & tl(\widehat{[\![e]\!]}\rho)
\end{array}
$$

$$
\begin{array}{l}
[\![\mathsf{node}\ N(x_1, \widehat{\ldots}\, x_k)\{x_j = e_j\}]\!] = \\
\qquad \lambda \widehat{f}_1, \ldots, \widehat{f}_k . \bigsqcup_{i=0}^{\infty} \widehat{F}^i(\widehat{f}_1, \ldots, \widehat{f}_k, [\infty], \ldots, [\infty])
\end{array}
$$

Figure 2: Clock analysis by abstract interpretation

5 Synchronisation

The most common operation on clocks is the synchronisation of two clocks. Synchronisation in the standard language was expressed using greatest lower bounds of sequences, extended to lists. We shall now define an abstract counterpart to this operator over the abstract lattice of clocks.

Definition 5.1 *Let $\kappa_1, \kappa_2 \in Clock$ and $\widehat{\kappa_1}, \widehat{\kappa_2} \in \widehat{Clock}$. Then an abstract synchronisation operator*

$$\S : \widehat{Clock} \times \widehat{Clock} \rightarrow \widehat{Clock}$$

is sound *if for all $f \in$* Flow*:*

$$\kappa_1 \sim_f \widehat{\kappa_1} \quad \wedge \quad \kappa_2 \sim_f \widehat{\kappa_2} \quad \Rightarrow \quad \kappa_1 \sqcap \kappa_2 \sim_f \widehat{\kappa_1} \ \S\ \widehat{\kappa_2}.$$

For the lattice $\mathbf{Ord}^\top$, synchronisation of two abstract elements n, m is obtained by taking the minimum of n and m since $\top^n \sqcap \top^m = \top^{min(n,m)}$. Synchronising a symbolic clock x_j with itself results in the x_j itself, since $\sqcap$ is idempotent, whereas synchronisation between two different symbolic clocks yields $\top$ since we don't have access to the actual clocks they represent. These are the main points in the proof of soundness of the following abstract synchronisation operator:

Example 5.2 *The abstract synchronisation operator*

$$\S : \mathbf{Ord}^\top[X] \times \mathbf{Ord}^\top[X] \rightarrow \mathbf{Ord}^\top[X]$$

is defined to be the commutative extension of

$$
\begin{array}{lcll}
n \ \S\ m & = & min(n, m) & \\
n \ \S\ x & = & \top & \\
x_i \ \S\ x_i & = & x_i & \\
x_i \ \S\ x_j & = & \top & \text{if } i \neq j \\
\_ \ \S\ \top & = & \top &
\end{array}
$$

It is extended to $\widehat{Clock} = List(\omega^\top[X])$ by elementwise application.

The synchronisation of general clocks in the standard semantics was defined by elementwise application of $\sqcap$ on simple clocks. The soundness of $\S$ on lists of simple clocks is therefore automatic from the soundness of $\S$ on simple clocks when $\S$ is extended to lists by elementwise application.

6 Abstract interpretation of $\mathcal{L}$

We shall now formulate an abstract interpretation for calculating approximations of clocks. It is assumed that the transformation from Section 4.2 has been performed so that all samplings in the analysed program is of form $e \mathsf{when} x$. The interpretation will be generic in the abstract clock domains and the abstract synchronisation operator. We prove that it is a correct approximation of the $\mathcal{L}$ semantics from Section 3.2. The abstract interpretation of a node with k input variables and n equations

$$\widehat{[\![\ ]\!]} : Node \rightarrow (\widehat{Clock})^k \rightarrow (\widehat{Clock})^n$$

is then induced from the standard semantics using the abstract interpretation of expressions. A node

$$\mathsf{node}\ N(x_1, \ldots x_k)\{x_j = e_j\}_{j=k+1}^n$$

defines a function $\widehat{F}$ on tuples of abstract flows by

$$\widehat{F}(\widehat{f}_1, \ldots, \widehat{f}_n) = (\widehat{f}_1, \ldots, \widehat{f}_k, \widehat{[\![e_{k+1}]\!]}\widehat{\rho}, \ldots, \widehat{[\![e_n]\!]}\widehat{\rho})$$

where $\widehat{\rho}(x_j) = \widehat{f}_j$ for $j = 1, \ldots, n$. This iterator is used to define the abstract interpretation of nodes:

$$
\begin{array}{l}
[\![\mathsf{node}\ N(x_1, \widehat{\ldots}\, x_k)\{x_j = e_j\}]\!] = \\
\qquad \lambda \widehat{f}_1, \ldots, \widehat{f}_k . \bigsqcup_{i=0}^{\infty} \widehat{F}^i(\widehat{f}_1, \ldots, \widehat{f}_k, [\infty], \ldots, [\infty])
\end{array}
$$

The correctness of this abstract interpretation is proved in three steps. First we show that the concrete and abstract interpretations of a single expression correspond provided that the environments in which the expression is interpreted do. Then we show that concrete and abstract iteration in the definition of the interpretations of a node corresponds and this is then used to show that the abstract interpretation of a node correctly approximates the standard semantics of the node.

Lemma 6.1 *Let e be an $\mathcal{L}$-expression with free variables $x_1, \ldots, x_n \in X$. Assume $f \in Flow^n$ and $l \in Clock^n$ such that $\chi(f_j) \sim_f l_j$. Let $\rho = [x_j \mapsto f_j]$ and $\widehat{(\rho)} = [x_j \mapsto l_j]$ be the concrete and the abstract environment f and l. Then*

$$\chi(\llbracket e \rrbracket \rho) \sim_f \widehat{\llbracket e \rrbracket} \widehat{\rho}.$$

Proof By induction over the structure of e.

$e = k$. Follows from $\chi(k^\infty, [\top^\infty]) = [\top^\infty] \sim_f [\omega]$.

$e = x_j$. Follows from the assumption that

$$\chi(\rho(x_j)) = \chi(f_j) \sim_f l_j = \widehat{\llbracket x_j \rrbracket} \widehat{\rho}.$$

$e = e_1 \; op \; e_2$. Here op ranges over the pre-defined operators $+, and, \ldots$ and the initialisation operator $\rightarrow$. The correctness of this is a consequence of the correctness of the abstract synchronisation operator §.

$e = \mathsf{pre}(e)$. Immediate, since

$$\chi(\llbracket \mathsf{pre}(e) \rrbracket \rho) = \chi(\llbracket e \rrbracket \rho) \sim_f \widehat{\llbracket e \rrbracket} \widehat{\rho} = \widehat{\llbracket \mathsf{pre}(e) \rrbracket} \widehat{\rho}.$$

$e = e \; \mathsf{when} \; x_j$. By induction hypothesis, $\chi(\llbracket e \rrbracket \rho) \sim_f (\widehat{\llbracket e \rrbracket} \widehat{\rho})$. Similarly,

$$\chi(\llbracket x_j \rrbracket \rho) = \chi(\rho(x_j)) \;\sim_f\; (\widehat{\rho}(x_j)) = (\widehat{\llbracket x_j \rrbracket} \widehat{\rho}).$$

Since $\rho(x_j) = f_j$ and $True(|f_j|) \;\sim_f\; x_j$ we have that

$$True(|\rho(x_j)|) \;\sim_f\; x_j.$$

Together these two facts prove that

$$\begin{gathered} True(|\rho(x_j)|) :: (\chi(\llbracket e \rrbracket \rho) \sqcap \chi(\llbracket x_j \rrbracket \rho)) \\ \sim_f \\ x_i :: (\widehat{\llbracket e \rrbracket} \widehat{\rho} \; \S \; \widehat{\llbracket x_j \rrbracket} \widehat{\rho}). \end{gathered}$$

$e = \mathsf{current}(e)$. Since $\sim_f$ on lists means that the two lists are related elementwise it is immediate that

$$\begin{gathered} \chi(\llbracket \mathsf{current}(e) \rrbracket \rho) = tl(\chi(\llbracket e \rrbracket \rho)) \\ \sim_f \\ tl(\widehat{\llbracket e \rrbracket} \widehat{\rho})) = \widehat{\llbracket \mathsf{current}(e) \rrbracket} \widehat{\rho}. \end{gathered}$$

when $\chi(\llbracket e \rrbracket \rho) \sim_f \widehat{\llbracket e \rrbracket} \widehat{\rho})$. □

This result is now extended to cover the concrete and abstract iterators F and $\widehat{F}$ induced by a set of equations over a given input flow. With F and $\widehat{F}$ denoting the iterator as above we get

Lemma 6.2 *Let $f \in Flow^n, \widehat{f} \in \widehat{Clock}^n$ and assume that $f \; \gamma \; \widehat{f}$. For all $j = 1, \ldots, n$:*

$$\chi(F(f)_j) \sim_f \widehat{F}(\widehat{f})_j.$$

Proof Using Lemma 6.1 we immediately have:

$$\chi(F(f)_j) = \chi(\llbracket e_j \rrbracket \rho) \sim_f \widehat{\llbracket e_j \rrbracket} \widehat{\rho} = \widehat{F}(\widehat{f})_j.$$

□

The final part of the correctness proof for the clock analysis amounts to showing that the fixed point semantics of a node defined by the abstract interpretation is a correct approximation of its standard semantics. For a node with k input variables we abbreviate the initial values of the iterations as follows:

$$\begin{aligned} init &= (in_1, \ldots, in_k, (\epsilon, [\top^\infty]), \ldots, (\epsilon, [\top^\infty])) \\ \widehat{init} &= (\widehat{in}_1, \ldots, \widehat{in}_k, [\infty], \ldots, [\infty]). \end{aligned}$$

The correctness then follows from the following theorem:

Theorem 6.3 (Correctness) *Assume that $in_1 \sim \widehat{in}_1, \ldots, in_p \sim \widehat{in}_p$. Then*

$$\bigsqcup_{i=0}^{\infty} F^i(init) \;\; \gamma \;\; \bigsqcup_{i=0}^{\infty} \widehat{F}^i(\widehat{init}).$$

Proof The assumption together with the fact that $\top^\infty \sim \infty$ yields that the initial values of the iterations are related by $\sim$ and therefore by any $\sim_f$, in particular $\sim_{init}$. Writing F^i for $F^i(init)$ and $\widehat{F}^i$ for $\widehat{F}^i(init)$ we obtain that for any $i \in \mathcal{N}$ and $j \in 1, \ldots, n$,

$$F_j^{i+1} \sim_{F^i} \widehat{F}_j^{i+1}$$

by $i+1$ applications of Lemma 6.2. Since

$$F^\infty \equiv \bigsqcup_{i=0}^{\infty} F^{i+1} = \bigsqcup_{i=0}^{\infty} F^i$$

we can by Lemma 4.7 conclude that

$$F_j^\infty \sim_{F^\infty} \widehat{F}_j^\infty$$

i.e., that $F^\infty \; \gamma \; \widehat{F}^\infty$. □

As an example of how the analysis works we calculate the clocks in the monitor example from Section 2.

Example 6.4 The analysis of the monitor

$$\begin{aligned} &\mathsf{node}(Sec : int, In : int) \\ &\quad NewMin \;=\; (Sec \;\mathsf{mod}\; 60 = 0) \\ &\quad InS \;=\; In \;\mathsf{when}\; NewMin \\ &\quad Alarm \;=\; |\mathsf{current}(InS) - In| > Critical \end{aligned}$$

gives rise to the fixed point iteration shown below. The input flows Sec and In are both on the basic clock $[\infty]$.

0.	*Sec*	*In*	*NewMin*	*InS*	*Alarm*
1.	$[\infty]$	$[\infty]$	$[\infty]$	$[\infty]$	$[\infty]$
2.	$[\infty]$	$[\infty]$	$[\infty]$	$[NewMin, \infty]$	$[]$
3.	$[\infty]$	$[\infty]$	$[\infty]$	$[NewMin, \infty]$	$[\infty]$
4.	$[\infty]$	$[\infty]$	$[\infty]$	$[NewMin, \infty]$	$[\infty]$

and a fixed point is reached.

Rules for expressions

$$\Gamma[x:\kappa] \vdash x : \kappa \qquad \Gamma \vdash k : [\infty]$$

$$\frac{\Gamma \vdash e : \kappa}{\Gamma \vdash \mathsf{pre}(e) : \kappa}$$

$$\frac{\Gamma \vdash e_1 : \kappa \quad \Gamma \vdash e_2 : \kappa}{\Gamma \vdash e_1 \; op \; e_2 : \kappa} \qquad \frac{\Gamma \vdash e_1 : \kappa \quad \Gamma \vdash x : \kappa}{\Gamma \vdash e_1 \; \mathsf{when} \; x : x :: \kappa}$$

$$\frac{\Gamma \vdash e : \tau :: \kappa}{\mathsf{current}(e) : \kappa}$$

Rule for nodes

$$\frac{[x_1:\kappa_1,\ldots,x_n:\kappa_n] \vdash e_j : \kappa_j \quad \text{for } j = k+1,\ldots,n}{[x_1:\kappa_1,\ldots,x_k:\kappa_k \vdash \mathsf{node}\;(x_1,\ldots,x_k).\{x_j = e_j\} : (\kappa_1,\ldots,\kappa_n)}$$

Figure 3: Monomorphic clock analysis.

From the fixed point so obtained we see that the flows *NewMin* and *Alarm* are on the basic clock and that the clock of *InS* is defined by the flow of *NewMin*. All flows have thus been assigned a unique clock and we conclude that the program is well synchronised.

In general, a clock analysis calculates a superset of the possible clocks that the flows in a program can have. Depending on the definition of concrete language being analysed, this information will then lead us to reject or accept a program. For example, the LUSTRE language insists on flows having a unique clock hence all the supersets must be singletons for the program to be accepted. In Section 8 we present the SIGNAL language where a less restrictive approach is adopted.

7 A type system for clock analysis

The clock analysis presented in the previous section is less strict than existing analyses of LUSTRE in the sense that it will assign a non-$\top$ clock to more expressions than the LUSTRE analysers. This is due to the fact that our synchronisation operator will synchronise two clocks $\widehat{\kappa_1}$ and $\widehat{\kappa_2}$ to their greatest lower bound whereas LUSTRE insists on the clocks being equal for them to synchronise.

In this section we shall define a type system for verifying that a node of equations are well synchronised in the LUSTRE sense. The type system presented here is an equivalent of the abstract interpretation defined by Bergerand in [Ber86]: if the abstract interpretation of a node assigns non-$\top$ clocks to all expressions then the type system can prove that the node is well synchronised. We have chosen to use a type system presentation of the analysis because it provides a simple way of specifying an analysis and because it leads naturally to consider a notion of polymorphic clock which we shall use in the next section to discuss SIGNAL and non-determinism.

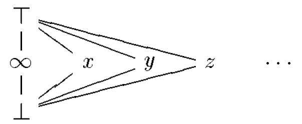
As our "types" we can use the elements of any lattice built using the constructions of Section 4 with the added requirement that the lattice for simple clocks must include ∞ describing the clock that ticks at all instants. Synchronisation is now expressed in the deduction rules as a requirement that certain sub-expressions of an expression must have the same clocks for a deduction to be valid as in the rule

$$\frac{\Gamma \vdash e_1 : \kappa \quad \Gamma \vdash e_2 : \kappa}{\Gamma \vdash e_1 \; op \; e_2 : \kappa}$$

for built-in operations. The rules defining the new clock analysis is shown in Figure 3.

The analysis defined by Bergerand essentially corresponds to the type system obtained by using the lattice $\omega^{\top}$ from Example 4.2. However, the stronger requirements on synchronisation enables a very compact representation of a tuple of such clocks. Since e and x are required to have the same clock κ in the expression e when x we can represent the resulting clock $x :: \kappa$ simply as x, knowing that κ can be retrieved as the clock of variable x.

This coding of lists of clocks through the environment means that the analysis can now be performed over a much reduced lattice. The lattice *HOR* used in [Ber86] has the form



where $x, y, z, \ldots$ are the variables of the program. One small restriction imposed by this lattice is that it is no longer possible to represent clocks such as $[\infty, \infty]$ for flows on the basic clock that has been sampled on a signal that was true all the time, Since this kind of clocks cannot be created by the analysis but only be furnished as clocks of input variables this is of minor importance.

8 Non-determinism and polymorphism

This section relates the analyses seen so far to the clock analysis of SIGNAL. Our aim is not to develop a new analysis for SIGNAL, but rather—by using the framework of type systems—to clarify the relationship between analyses that so far have been presented in rather different ways.

One feature of LUSTRE is that it is deterministic. As a consequence of this it is required that the clock of an expression in a program is determined by the expression itself and not by the context in which the expression occurs. This means that a counter like

$$C \quad = \quad 0 \rightarrow 1 + \mathsf{pre}(C)$$

is deemed to be on the basic clock $[\top^{\infty}]$ and must be slowed down by sampling in order to synchronise with clocks other than $[\top^{\infty}]$. The philosophy behind the design of the SIGNAL language is quite different: a flow has its own clock and this clock is only constrained by the interactions that the flow makes with other flows. A counter like C above will

Rules for expressions

$$\emptyset \vdash x : \alpha_x$$

$$\emptyset \vdash k : \alpha \qquad (\alpha \text{ a fresh clock variable})$$

$$\frac{C \vdash e : \kappa}{C \vdash \mathsf{pre}(e) : \kappa}$$

$$\frac{C_1 \vdash e_1 : \kappa_1 \quad C_2 \vdash e_2 : \kappa_2}{C_1 \cup C_2 \cup \{\alpha = \kappa_1, \alpha = \kappa_2\} \vdash e_1 \ op \ e_2 : \alpha} \qquad (\alpha \text{ a fresh clock variable})$$

$$\frac{C \vdash e_1 : \kappa}{C \vdash e_1 \ \mathsf{when} \ x : \kappa \cap x} \qquad \frac{C_1 \vdash e_1 : \kappa_1 \quad C_2 \vdash e_2 : \kappa_2}{C_1 \cup C_2 \vdash e_1 \ \mathsf{default} \ e_2 : \kappa_1 \cup \kappa_2}$$

Rule for nodes

$$\frac{C_j \vdash e_j : \kappa_j \quad \text{for } j = k+1, \ldots, n}{C_{k+1} \cup \ldots \cup C_n \cup \{\alpha_{x_i} = \kappa_i\}_{i=1}^{n} \vdash \mathsf{node}\ (x_1, \ldots, x_k).\{x_j = e_j\} : (\kappa_1, \ldots, \kappa_n)}$$

Figure 4: Constraint system for clock analysis.

therefore not be assigned a clock until it interacts with a flow with a well-defined clock. Synchronisation is forced by the built-in operators like $+$ and by a special program directive $\mathsf{synchro}(x, y)$ which specifies that the flows of x and y be synchronised. Clocks in SIGNAL can therefore be viewed as equivalence relations on flows [LB86] such that two flows are related if they must be on the same clock.

Sampling in SIGNAL is somewhat different from LUSTRE sampling in that e and x are not required to have equal clocks; rather e when x produces output exactly when e and x ticks and x carries the value true. This means that, for a given boolean flow *In*, the flow of *Out* in

$$\begin{array}{l} \mathsf{node}\ (In : \mathsf{bool}) \\ \quad C \quad = \quad 0 \rightarrow 1 + \mathsf{pre}(C) \\ \quad Out \quad = \quad C \ \mathsf{when} \ In \end{array}$$

can be any subsequence of the natural numbers since C produces these numbers at a rate that is independent of that of b's.

In one aspect SIGNAL is simpler than LUSTRE: it doesn't have the over-sampling operator current. The simplifies the clock analysis considerably since we are no longer obliged to keep a sampling record of a clock hence the analysis can be done with simple clocks only. On the other hand, SIGNAL has a merge construction

$$e_1 \ \mathsf{default} \ e_2$$

that will merge the flows of e_1 and e_2 with priority to e_1. The clock of e_1 default e_2 is the union of the clocks of e_1 and e_2 and the analysis must deal with this new operation on clocks.

8.1 A constraint system

In order to accommodate clocks that can vary according to the contexts we shall make use of "polymorphic clock variables" akin to the polymorphic types known from other languages such as ML [Mil78]. An expression with a polymorphic clock can be synchronised with any other expression of arbitrary clock including another polymorphic clock. The synchronisation results in the *constraint* that these two clocks must be equal. We present a system for inferring the set of synchronisation constraints that must be satisfied for a node to be well synchronised. The system is inspired by the use of constraint solving in analyses of strictness and binding-time properties of functional languages [Hen91, Amt94, HM94]

The constraint system operates on judgments of the form

$$C \vdash e : \kappa$$

where C is a set of constraints, *i.e.*, equations between clock expressions, that must be satisfied for e to have clock κ.

As before, the set of clock "types" contains a set of simple clocks and the set of program variables. A variable x is still used to represent the clock defined by the flow of x. These types are enriched with an infinite set of clock variables which we assume to include a variable α_x for each program variable x. To deal with the new sampling and merging operators we define two binary type constructors, $\cap$ and $\cup$, leading to the following set of clocks:

$$\begin{array}{rcl} \kappa & ::= & \mathit{SimpleClock} \mid x \mid \alpha \mid \alpha_x \\ & \mid & \kappa \cup \kappa \mid \kappa \cap \kappa \end{array}$$

The constraint system works as follows. A variable is assigned its "own" clock variable κ_x every time it is encountered. Every time this variable is involved in a synchronisation a constraint on κ_x will be produced. For constants we adopt a more liberal viewpoint: a constant flow can synchronise with any other flow since it can deliver that constant value at any clock rate. Furthermore different occurrences of the same constant need not have the same clock hence every occurrence is assigned a fresh clock variable. Neither variables nor constants produce any constraints in themselves.

Constraints are generated by the synchronous operators $+, \rightarrow, \ldots$: if $C_1 \vdash e_1 : \kappa_1$ and $C_2 \vdash e_2 : \kappa_2$ then $e_1 \ op \ e_2$ will be assigned a fresh clock variable α under the constraints

$$C_1 \cup C_2 \cup \{\alpha = \kappa_1 = \kappa_2\}.$$

As already mentioned, sampling doesn't require synchronisation but the resulting clock e when x is given by the expression $\kappa \cap x$ where κ is the clock of e. Likewise, e_1 default e_2 unions the set of constraints generated by e_1 and e_2 and has as clock $\kappa_1 \cup \kappa_2$.

Finally, a set of equations $\{x_i = e_i\}_{i \in I}$ is analysed by deriving a judgment $C_i \vdash e_i : \kappa_i$ for each equation and then forming the set of constraints

$$C_1 \cup \ldots \cup C_n \cup \{\alpha_{x_i} = \kappa_i\}_{i \in I}.$$

The rules appear in Figure 4.

As an example, consider the non-deterministic node from above:

$$\begin{array}{lcl} \multicolumn{3}{l}{\mathsf{node}\ (In : \mathsf{bool})} \\ C & = & 0 \rightarrow 1 + \mathsf{pre}(C) \\ Out & = & C\ \mathsf{when}\ In \end{array}$$

Letting α_0 and α_1 denote the clocks associated to the occurrences of the constants 0 and 1, the analysis of this node yields the following set of constraints (after elimination of some intermediary clock variables)

$$\alpha_0 = \alpha_1 = \alpha_C$$
$$\alpha_{Out} = \alpha_C \cap In$$

We see that nothing constrains the clock of flow C which can therefore proceed at its own pace as explained above.

Changing the example slightly, we assume given a synchronous operator $stamp(x, y)$ that will stamp the number x onto datum y. The following node will number the input arriving on In in the order it arrives.

$$\begin{array}{lcl} \multicolumn{3}{l}{\mathsf{node}\ (In : \mathsf{bool})} \\ C & = & 0 \rightarrow 1 + \mathsf{pre}(C) \\ Out & = & stamp(C, In) \end{array}$$

The constraints generated for this node are

$$\alpha_0 = \alpha_1 = \alpha_C$$
$$\alpha_{Out} = \alpha_C = \alpha_{In}$$

which therefore fixes all flows to be on the clock of the input In. This implies that the counter C will follow the rhythm of the input and the data arriving will be numbered correctly.

Once a set of constraints has been obtained we are left with the problem of solving it or at least finding out whether it admits a solution. We shall not enter into the details of how this is done (see [LB86],[GLB87] or [BLL92]) but just mention that the SIGNAL system employs an ingenious method for solving sets of constraints based on solving a set of equations over the finite field $F_3 = Z/Z_3$. The three elements in F_3 encodes the properties "absent", "present and have value *true*" and "present and have value *true*" for a boolean flow which allows to take a certain amount of information of the value of the flows into account when analysing the clocks. If the constraints have a solution such that all clocks are different from "absent" the original system is well synchronised but, as usual with approximate methods, the other implication is not true.

9 Conclusion

We have presented a denotational semantics of a synchronous dataflow language and shown how this semantics can be used to define and prove the correctness of clock analyses for this language. Our semantic framework is less general than that of [BLSS92] but its simplicity has made a correctness proof of the analysis feasible. To the best of our knowledge this is the first correctness proof for a LUSTRE-like clock analysis and thus fills a gap left open in the works of Plaice and Bergerand. Moreover, by formalising clock analysis in terms of the well-understood framework of abstract interpretation we hope to have clarified how to design other analyses by varying the domain of abstract clocks to suit the desired level of precision.

We then gave a type system formulation of the clock analysis. This type system allows to prove formally that a given assignment of clocks to flows is correct and plays a rôle similar to that of type checking in explicitly typed languages. The system is only intended for checking that a clock assignment for a given program is correct. An inference system for finding clocks is presented by Plaice [Pla88] who provides a system for full LUSTRE. The added capability of inferring clocks complicates the system since a kind of fixed iteration has to be encoded in the system.

The type system was then augmented with *clock variables* to arrive at a polymorphic clock analysis. This analysis provides a nice way of presenting the clock analysis for the SIGNAL language where the clock of an expression can be left undetermined if the flow does not synchronise with other flows. We sketched how the analysis of a program gives rise to a set of constraints on the clocks of a program. It is our hope and belief that the type system presentation of the various analyses will provide a way of understanding the similarities and differences between these analyses that so far have been presented in rather different ways.

Our clock analysis is formulated as an abstract interpretation using the lattice construction $L[X]$ where a lattice L of semantic properties is augmented with syntactic components from the program being analysed, here the variables appearing in the program. With this construction we can capture a certain amount of relational properties [JM81, CC79] of the semantics of a program *i.e.*, that two expressions have the same (unspecified) property. Further work will show whether this construction can be of use for other kind of analyses *e.g.*, as a vehicle for integrating methods based on fixed point iteration with symbolic calculations. Also, it should be possible to combine the two kind of information to obtain properties such as "this value has property $l \in L$ and is determined by variable x" by decorating the elements of L with variable names, rather than keeping them disjoint as in $L[X]$.

Finally, we just point out one interesting link with the use of $\leftrightarrow$ in the domain *Prop* from analysis of logic programs [CFW91, MSJ94] Here the formula $x \leftrightarrow y$ is used to represent that x and y have the same abstract, but unspecified, property. Since *Prop* only works over the two-point lattice our approach offers a generalisation of $\leftrightarrow$ since it works for arbitrary lattices. However, it should be noted that *Prop* contain other formulae, such as $x \rightarrow y$, meaning that if x satisfies a property then so does y—this is not present in our construction.

References

[Amt94] Torben Amtoft. Local type reconstruction by means of symbolic fixed point iteration. In *Proc. of 5th European Symposium on Programming, Edinburgh, U.K.*, pages 43–57. Springer Verlag LNCS 788, 1994.

[BB91] A. Benveniste and G. Berry, editors. *Another look at real-time programming. Special issue of Proceedings of the IEEE, vol. 79, no. 9.* IEEE Press, September 1991.

[Ber86] J.-L. Bergerand. *LUSTRE: un langage déclaratif pour le temps reel.* PhD thesis, Institut national polytechnique de Grenoble, 1986.

[BLL92] A. Benveniste, M. Le Borgne, and P. Le Guernic. Signal as a model for real-time and hybrid systems. In B. Krieg-Brückner, editor, *Proc. of 4th European Symposium on Programming*, LNCS vol. 582, pages 20–38. Springer Verlag, Berlin, 1992.

[BLSS92] A. Benveniste, P. Le Guernic, Y. Sorel, and M. Sorine. A denotational theory of synchronous reactive systems. *Information and Computation*, 99(2):192–230, 1992.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixpoints. In *Proc. of 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, 1977. ACM Press, New York.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. 6th ACM Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York.

[CFW91] A. Cortesi, G. Filè, and W. Winsborough. *Prop* revisited: Propositional formula as abstract domain for groundness analysis. In *Proc. of 6th Symposium on Logic in Computer Science*, pages 322–327. IEEE Computer Society Press, 1991.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Proc. of 14th ACM Conf. on Principles of Programming Languages*, pages 178–188. ACM Press, 1987.

[GLB87] T. Gautier, P. Le Guernic, and L. Besnard. Signal : A declarative language for synchronous programming of real-time systems. In G. Kahn, editor, *Proc. of 3. ACM Conf. on Functional Programming and Computer Architecture*, pages 257–277. Springer LNCS vol. 274, 1987.

[Hal93] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Publishers, 1993.

[Hen91] F. Henglein. Efficient type inference for higher–order binding–time analysis. In J. Hughes, editor, *Proc. of 5th ACM Conference on Functional Programming Languages and Computer Architecture*, LNCS vol. 523. Springer Verlag, 1991.

[HM94] F. Henglein and C. Mossin. Polymorphic binding-time analysis. In D. Sannella, editor, *Proc. of 5th European Symposium on Programming*, LNCS vol. 788, pages 287–301. Springer Verlag, 1994.

[JM81] N.D. Jones and S.S. Muchnick. Complexity of flow analysis, inductive assertion synthesis and a language due to Dijkstra. In N. Jones and S. Muchnick, editors, *Program Flow Analysis: Theory and Applications.* Prentice-Hall, 1981.

[Kah74] G. Kahn. The semantics of a simple language for parallel processing. In *Information Processing 74. Proc. of IFIP Congres*, pages 471–475. North-Holland, 1974.

[KM77] G. Kahn and D. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processinhg 77. Proc. of IFIP Congres*, pages 993–998. North-Holland, 1977.

[LB86] P. Le Guernic and A. Benveniste. Real-time, synchronous data-flow programming: the language SIGNAL and its mathematical semantics. Rapports de recherche 533, IRISA/INRIA, Rennes, 1986.

[LGLL91] P. Le Guernic, T. Gauthier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[MSJ94] K. Marriott, H. Søndergaard, and N. Jones. Denotational abstract interpretation of logic programs. *ACM Trans. on Programming Languages and Systems*, 16(3):607–648, 1994.

[Pla88] J. A. Plaice. *Sémantique et compilation de LUSTRE, un langage déclaratif synchrone.* PhD thesis, Institut national polytechnique de Grenoble, 1988.