Preuves et Programmes Synchrones: a Survey[†]

Synchronous Programming Languages for Faust

Olivier Hermant
MINES ParisTech, PSL Research University

19 Décembre 2014

1 Introduction

This related work survey considers a panel of articles around synchronous programming languages, their design and uses. The goal is to extract information that can be potentially useful to the Faust programming language. In particular we will be concerned with :

— the synchronous model and the notion of clock ;

— the formal guarantees provided by synchronous languages, among which :
  - worse case execution time (WCET) computations ;
  - multi-clock systems, in particular in link with the multirate version of Faust ;
  - operational and denotational semantics ;
  - formalizations of (parts of) some synchronous languages in proof-assistants such as Coq, and especially the goals that we can reach : safety guarantees on programs, clearly specified semantics, other theorems that can be proved on the language (on its syntax, on its semantics,...).

— parallelization of tasks on more than one core, and distributed models of computing

The list of analyzed related works is obviously non exhaustive, given the wealth of the subject matter, the selection process being based on numerous discussions with experts in various institutions[1]. We will possibly look at

[†]This work has been supported by the Carnot M.I.N.E.S contract PPS, N^o 30660

1. In particular, we thank, among others, Yann Orlarey, Marc Pouzet, Emilio Gallego and Pierre Jouvelot for their help.

other papers in subsequent updates to this document. The ten papers considered here are [BJMP13, BBCP12, BT13, BP13, BH01b, BH01a, CMPP08, DGP08, EL03, MPP11, RRJ+14]. In the sequel, we assume that the reader is familiar with the theoretical pinnings of synchronous systems and has some knowledge about the Faust programming language.

This document is organized as follows. Sec. 2 introduces synchronous programming languages. Sec. 3 establishes a current picture of the state of the art, based on the review of several key papers, while providing insights on the impact this study can have on the Faust ecosystem. Sec. 4 contains all the details of the reviewed papers, and a synthesis of their possible use for Faust. We briefly conclude in Sec. 5.

2 Synchronous Programming Languages

Synchronous programming languages have had a very strong impact on critical software development, in particular in aeronautics and, even more, in the design of circuits (see the Esterel compiler or SCADE [PEB07]). They are based on (a priori) simple concepts we briefly recall here.

2.1 The Synchronous Paradigm

Synchronous programming languages have been designed to write programs that answer to stimuli coming from the outside environment or from other programs. For this, the approximation retained is that *the treatment of information is instantaneous* or, at the least, that one always has sufficient CPU time to react to stimuli as soon as they come without delay. This means that, seen from the outside, the answer time of a *reactive* (or synchronous) program can be considered to be immediate.

¿From within a given program, when a stimuli is received, a handler is triggered. In practice, this means that the signals are coming at a rate that allows the program to handle them in time. This assumption should, in theory, be verified by computing the Worse Case Execution Time (WCET) of each event handler program. WCET evaluation can be very precise, for instance by looking at assembly code, involving memory and scheduler models, etc. Controlling WCET imposes strong constraints on the available primitives : in particular, from this point of view, recursive functions or loops, with their difficult-to-predict completion times, are bad, so they are quite often forbidden. Once this timing constraint is globally ensured, time can be considered as a purely *logical*, ordering concept, and clocks do not have to be bound to any periodic (clock) function in the physical world, in contrast to Faust, for instance. Since one uses this zero-execution time approximation, computations are considered to be immediate and always take place between two (clock) instants, or ticks. Therefore the logical nature of clocks has not much importance and we can still speak about logical synchronization, timing and so on. Note that this (possibly) can be a problem for programs that use an internal model of physical time, such as control-command software.

In summary, abiding by the time model discussed above, everything occurs in a synchronous fashion, and task handling can be seem as real parallelism : reactions are triggered by stimuli that can occur concurrently, and only on the instants determined by the clock.

2.2 Is Faust Synchronous ?

In the sense of the previous section, Faust is a synchronous language : the stimuli received (the audio samples) are clocked. The clock can still be understood as a logical one even if the clock ticks are physically periodic (usually, at a rate of 44kHz). Yet, it has to be noticed that the clock model of Faust does not need all the features of the clock models of synchronous languages, such as clock polymorphism, aperiodic clocks or ultimately periodic clocks (see Sec. 3 below for more details).

The "synchronous programming language" assumption discussed above, that the execution time of a program can be considered as zero with respect to the signal period, is also made in Faust. The WCET can be approximated by the length of the longest path from the source to the output, since there is no immediate feedback loop when computing a given audio sample. Indeed, an admissible approximation to compute the length (WCET) of the longest path would be to disconnect all "wires" at the place where a delay operator appears, and then compute the path on the resulting oriented graph, topologically sorted.

3 State of the Art

This section, which builds upon our analysis of the related work detailed in Sec. 4, provides a global analysis of the key concepts addressed by the synchronous programming community. We end with a discussion on the application of this overview to the case of Faust.

3.1 The Time Model : Clocks

As already said, clocks are logical, and not bound to physical time. A clock is therefore an infinite sequence of instants, or ticks. Events can have different clocks, so one must be able to speak about multiple clocks. Since one also needs to reason about synchronicity, the succession of instants of different clocks must be placed with respect to each other.

This calls for the following formulation. One assumes a *root clock*, still unrelated to physical time, and then defines the other clocks on top of it : a tick of a clock is represented by a `1` at the corresponding instant of the root clock, while a `0` represents the absence of a tick at this particular instant. For instance, here are two examples of clocks :

```
clk1 := 01001010100101010010l...
clk2 := 10101010101010101010l...
```

Usually, one is interested in clocks that follow some regularity pattern, among which *periodicity* is of paramount importance : one needs to design communication between different processes, and, since the clocks are infinite streams, knowing whether two processes that are sequenced according to different clocks can be synchronized through bounded buffers, or not, is of key importance. With periodic clocks, a fixed pattern is repeated infinitely often. To represent them, one can reuse the facilities offered by regular expressions, for instance writing `clk2 := (10)*` for the clock above.

An extension of periodicity is *ultimate periodicity*, meaning that, before enjoying its periodic behavior, a clock can have a finite number of lawless instants. This also fits into regular expressions very well. For instance `clk1 := 0(1001010)*`. This kind of expression allows ones to :

— manipulate clocks in the programming language as first-class citizens [CMPP08, MPP11],

— in particular, negate clocks, prefix them or filter a clock on another one ;

— express *clock polymorphism* for a process, in which case, for instance, one can declare (or infer, with clock inference) that a process must have the clock type α filtered by `(01)*`, whatever a particular clock α (bound to another process) is ;

— check causality (i.e., if two clocks `clk1` and `clk2` are synchronous, then the instants of `clk1` must come before the instants of `clk2`) with notions such as clock precedence or no instantaneous feedback ;

— express and check compatibility by means of a *subtyping relation* ;

— and, once compatibility is establish, compute the minimal buffer sizes between processes, through an abstraction relation (for performance reasons rather than decidability) [CMPP08, MPP11, BT13].

Lastly, one can want to take into account aperiodic clocks [BT13], for events produced by the physical world, for instance. Even if they cannot be formalized as precisely as periodic clocks, one can still describe some of their characteristics, for instance for "*a clock that produces one event every 10 instants, in mean*" or "*that produces exactly one event, randomly in a time frame of size 10, that repeats every 10 instants*", or even "*a Poisson process*".

It is also possible to describe the available resources [BT13], in terms of computation power and memory space, wrt to the logical clock, so as to compute precisely WCET and to check that we do not exceed the computation capacities. While this does still not correspond to true physical time, this goes quite close to it.

3.2 Semantics and its Formalization

The goal of a formal semantics is to describe unambiguously the behavior of programs through denotational or operational descriptions. This allows for (quasi-)deterministic predictability of any program, under some implicit or explicit assumptions (for instance, that the rate of events do not exceed the WCET), which is a good thing to ensure.

Going one step further, such an endeavor also allows to express formally properties on programs or on the language itself and, possibly, prove them. This is particularly the case when by "formally" one means "via a proof-assistant". This is the case of [BH01b] and [BH01a], which express the semantics of Lucid-synchrone in Coq, and of [RRJ+14].

The latter case ([RRJ+14]) is the most advanced one, with a whole tool chain. Properties to prove are expressed in a small Domain Specific Language (DSL) which is used to specify annotations to the targeted language (called REFLEX). The REFLEX primitives have properties defined through pre- and post-conditions. The scope of the annotations added to REFLEX code are the *traces* generated by the execution of programs, which are abstractions over programs themselves and whose adequacy to real program behaviors is verified. Note that the REFLEX interpreter is written using the Ynot library, which already provides a high degree of formalization. Since the properties are stated as annotations, an automatic machinery generates proof scripts for Coq, making use of a specific Coq library of basic results on the language, in particular on the properties stated on the language's primitives mentioned above.

Another interesting path regarding formalization is the introduction of DSLs for music expressed in a synchronous language. At the cost of compiling towards such a language (and so, maybe losing in efficiency), one benefits of all the guarantees and formalization results that the target synchronous language has led to. This is the case of [BJMP13], where Antescofo is expressed in the ReactiveML synchronous language.

3.3 The Continuous Case

The so-called "hybrid languages" correspond to synchronous language where some continuous primitives are added, under the form of *ordinary differential equations* [BP13, BBCP12]. Such an extension is usefull when one wants to model, inside the language itself, the external environment, which is by nature continuous. Then, it becomes possible to express directly the stimuli produced, which, without loss of generality, can be defined as zero crossings of continuous, differentiable functions. Some restrictions are, of course, present : in particular, the interactions between the continuous and discrete parts follow a strict policy while causality is always enforced.

In the final running program, the continuous part is discarded, since at the end, one does not need to model the environment. But in the intermediate steps, the runs of a given program are coupled with a numerical solver of its differential equations, which handles the updates coming from the synchronous reactive part of the program, and computes the adequate function, until a zero is crossed, in which case an event is generated. Thus, the assumption of instant reactiveness of the discrete part of a program is still valid. A point of interest is that the semantics of such hybrid languages can be expressed by the means of non-standard analysis citeABenTBouBCaiMPou12, a nice application of advanced mathematics.

3.4 Multitarget Compilation

As already mentioned, synchronous languages react to external stimuli. It is therefore a natural idea to postulate that a *single* program can be executed on *several* different targets, typically devices in a multi-agent system. An additional difficulty is that those targets can be heterogeneous.

Several works are tackling this issue, which is not yet fully solved.

— The scheduling of tasks for execution on multiple processors, in the case of periodic and sporadic tasks, is discussed in [BT13], where precise computations of WCET and processor capacities are addressed, to check that the schedule is realizable.

— Syntactic extensions to specify the target (location) on which a piece of code should execute and to constrain the communications allowed between targets is discussed in [DGP08].

Another work on compilation is the one of [EL03], which focuses on the evaluation of boolean circuits that sport instantaneous feedback while being still deterministic. Determinism is reached through the use of a lattice for the values transferred on the wires. In this framework, one must evaluate a circuit several times, assuming that, at the first iteration, the values of the internal wires are *undefined*. This paper tackles the issue of optimizing the order of evaluation, so that one performs as few iterations as possible until a fixed point is reached.

3.5 Possible Applications to Faust

As already argued, Faust *is* a synchronous programming language, although it exhibits very peculiar characteristics. We review in this section how the progress made in the domain of general-purpose synchronous languages can be leveraged to the Faust audio case, as well.

Clocks. In the monorate case, Faust operates on one single, periodic clock. This is strongly related to hard real time. In the multirate case, the clocks are related by a least common multiple relation, which represents the root clock. The periodicity seems to be absolute. The clock calculi presented in the literature seem too powerful for the use that can be made in Faust. However some ideas, such as using abstract interpretation and computation of buffer sizes, can be driven by those works.

Execution model. The current model of execution in Faust is local. This could change according three points of view.

— Multicore compilation. Every computer now has multiple cores. Faust could take advantage of this and compile its different parts on different processors, so as to speed-up execution. A first attempt at using OpenMP and vector instructions is provided by the current Faust compiler.

— One step further. One could enlarge the idea of multicore compilation to include the use of various specific devices of a computer, such as sound processor or a GPU accelerator.

— Even a step further. In some musical performances, Faust has been used on the audience's mobile-connected devices, together with a master program. This idea can be taken to a higher level by introducing location primitives, so that this kind of application can be written in one single program.

Formalizing Faust. The REFLEX approach is very interesting on the long term. However, REFLEX is a toy language, and a similar work, automatizing all the proofs, may be very involved. First of all, we have to identify the kind of properties that would be interesting to show. Traces could for instance be the output streams produced by a Faust program.

Many other traits of the Faust environment also deserve to be formalized, such as typing rules or execution semantics, for instance. Mimicking the approach of compiling (or interpreting) Antescofo towards a synchronous language seems out of reach for Faust : the two languages are quite different and the efficiency considerations which form a core concern in Faust may be at odds. If such a translation was extremely natural for Antescofo, it would require much more work for Faust.

More extensions. Following the path taken by hybrid languages, it could be useful at the fringe, for instance, to introduce analog events in Faust (think for instance of analog sliders). Currently, sliders are sampled, as well as the rest.

Finally, note that Faust does not have any problem with causality, since the feedback loop is systematically, by construction, guarded by a delay. However, striving to decompose this operation in more elementary ones, e.g., in order to optimize it, could lead us to solve several problems, such as those raised by [EL03].

4 Papers' Detailed Analysis

We list below the research papers that were looked at in this survey. For each of them, we give its title, authors' names, publication venue and abstract ; this is followed by our own synthesis and a discussion addressing the potential benefits (or lack thereof) we see in this line of work w.r.t. to the research currently linked to the Faust programming language.

4.1 Paper [BT13]

Title	: Design of Safety-Critical Java Level 1 Applications Using Affine Abstract Clocks
Authors	: Adnan Bouakaz and Jean-Pierre Talpin
Published in	: 16th International Workshop on Software and Compilers for Embedded Systems

Abstract : Safety-critical Java (SCJ) is designed to enable development of applications that are amenable to certification under safety-critical standards. However, its shared-memory concurrency model causes several problems such as data races, deadlocks, and priority inversion. We propose therefore a dataflow design model of SCJ applications in which periodic and aperiodic tasks communicate only through lock-free channels. We provide the necessary tools that compute scheduling parameters of tasks (i.e. periods, phases, priorities, etc) so that uniprocessor/multiprocessor preemptive fixed-priority schedulability is ensured and the throughput is maximized. Furthermore, the resulted schedule together with the computed channel sizes ensure underflow/overflow-free communications. The scheduling approach consists in constructing an abstract affine schedule of the dataflow graph and then concretizing it.

4.1.1 Synthesis

This paper is about scheduling the handling of aperiodic and periodic tasks, and the communications between them. The periodic tasks are linked to each other through affine clock-relations (each periodic task has its own clock). An algorithm to find the optimal buffer size and schedule, while avoiding under- and overflows, is discussed, both on a single processor and on multiple processors.

4.1.2 Discussion

It seems that, by simplifying the argument, one could apply the technique described here to make Faust compile on multiple processors by splitting a Faust program into several parts that communicate with each other through channels.

Faust is currently a purely periodic language, since it works at the sampling rate, which would represent the base rate, that is to say, in the language of the paper, $x(i) = 1$, for any $i \in \mathbb{N}^*$. One could nevertheless envision to add aperiodic tasks to Faust, to take into account manual user input for instance : instead of sampling the signal of a 0/1 switch into a periodic signal, which is quite costy, one could assign it to an aperiodic task.

Affine relations can be used to handle the multirate extension of Faust. In our case, those relations should be linear, since there is no “initialization problem". Yet, could one think about an extension of Faust where this could be the case ? Would there be any interesting use case ?.

4.2 Paper [DGP08]

Title	: A Type System for the Automatic Distribution of Higher-order Synchronous Dataflow Programs
Authors	: Gwenael Delaval, Alain Girault, and Marc Pouzet
Published in	: ACM International Conference on Languages, Compilers, and Tools for Embedded Systems

Abstract : We address the design of distributed systems with synchronous dataflow programming languages. As modular design entails handling both architecture and functional modularity, our first contribution is to extend an existing synchronous dataflow programming language with primitives allowing the description of a distributed architecture and the localization of some expressions onto some processors. We also present a distributed semantics to formalize the distributed execution of synchronous programs. Our second contribution is to provide a type system, in order to infer the localization of non-annotated values by means of type inference and to ensure, at compilation time, the consistency of the distribution. Our third contribution is to provide a type-directed projection operation to obtain automatically, from a centralized typed program, the local program to be executed by each computing resource. The type system as well as the automatic distribution mechanism has been fully implemented in the compiler of an existing synchronous data-flow programming language.

4.2.1 Synthesis

It is a ather theoretical paper which would deserver a very thorough analysis. It discusses an extension of higher-order functional synchronous languages (like Lustre, Lucid Synchrone, and so on) to make them able to split the execution of programs on multiple *locations* (without any further precision, so an instance of a location could be a sensor, a processor, etc.).

The approach retained is to let the user *specify explicitly* the location s/he wants a particular piece of code to be executed. This induces a type system that takes into account such spatial information. It is also possible to indicate which channels exist between each location, and to have a form of polymorphism *à la* ML for locations as well. This ends up with a set of location constraints that have to be solved.

A distributed operational semantics is defined, as well as *projection* operations that map to every location the suitable piece of code.

4.2.2 Discussion

Faust probably does not need such complex mechanisms that allow the user to mention explicitly where a particular piece of code has to be executed. In particular, this would not be useful for a mere multicore compilation, for all the job is most of the time better managed by the compiler. We can imagine that a use of this paradigm could be to make possible a *single* Faust program to run on *different* devices. The clear advantage of this compared to having different Faust programs would be the possibility to analyze this single program as a whole. Currently, this looks like a long-term goal.

4.3 Paper [BP13]

Title	: Zélus, a Synchronous Language with ODEs
Authors	: Timothy Bourke and Marc Pouzet
Published in	: In International Conference on Hybrid Systems : Computation and Control (HSCC 2013)

Abstract : Zélus is a new programming language for modeling systems that mix discrete logical time and continuous time behaviors. From a user's perspective, its main originality is to extend an existing Lustre-like synchronous language with Ordinary Differential Equations (ODEs). The extension is conservative : any synchronous program expressed as data-flow equations and hierarchical automata can be composed arbitrarily with ODEs in the same source code.
A dedicated type system and causality analysis ensure that all discrete changes are aligned with zero-crossing events so that no side effects or discontinuities occur during integration. Programs are statically scheduled and translated into sequential code which, by construction, runs in bounded time and space. Compilation is effected by source-to-source translation into a small synchronous subset which is processed by a standard synchronous compiler architecture. The resulting code is paired with an off-the-shelf numeric solver.
This experiment shows that it is possible to build a modeler for explicit hybrid systems à la Simulink/Stateflow on top of an existing synchronous language, using it both as a semantic basis and as a target for code generation.

4.3.1 Synthesis

This paper describes a synchronous language in which one can express a plant (the physical model) by means of continuous variables that follow laws determined by ODEs (ordinary differential equations). The interaction of those variables with the more usual clocked synchronous language is handled by the detection of *zero crossings*, that send signals (events) and trigger some reactions. This allows to make very realistic hybrid simulations (hybrid in the sense of clocked synchronous programs mixed with continuous variables) by introducing into the operational semantics of such a language external tools (numeric solvers) specialized into the integration of ODEs and that can detect zero crossings. Notice that the hypothesis that the synchronous program has always enough time to react quasi-instantly still holds.

4.3.2 Discussion

Integrating ODEs to be able to simulate the real-world environment seems very interesting but represents, we believe, a huge amount of work for a limited benefit in the case of Faust (audio signal).

4.4 Paper [BBCP12]

Title	: Non-Standard Semantics of Hybrid Systems Modelers
Authors	: Albert Benveniste, Timothy Bourke, Benoit Caillaud and Marc Pouzet
Published in	: Journal of Computer and System Sciences (JCSS)

Abstract : Hybrid system modelers have become a corner stone of complex embedded system development. Embedded systems include not only control components and software, but also physical devices. In this area, Simulink is a de facto standard design framework, and Modelica a new player. However, such tools raise several issues related to the lack of reproducibility of simulations (sensitivity to simulation parameters and to the choice of a simulation engine).
In this paper we propose using techniques from non-standard analysis to define a semantic domain for hybrid systems. Non-standard analysis is an extension of classical analysis in which infinitesimal (the ε and η in the celebrated generic sentence for all ε there exists η ... of college maths) can be manipulated as first class citizens. This approach allows us to define both a denotational semantics, a constructive semantics, and a Kahn Process Network semantics for hybrid systems, thus establishing simulation engines on a sound but flexible mathematical foundation. These semantics offer a clear distinction between the concerns of the numerical analyst (solving differential equations) and those of the computer scientist (generating execution schemes).
We also discuss a number of practical and fundamental issues in hybrid system modelers that give rise to non reproducibility of results, nondeterminism, and undesirable side effects. Of particular importance are cascaded mode changes (also called “zero-crossings” in the context of hybrid systems modelers).

4.4.1 Synthesis

This article, which also would deserve a more in-depth analysis of its own, presents different semantics for hybrid systems languages, the same as the ones overviewed in Section 4.4. This work goes mathematically very deep, and exhibits a semantics that makes use of non-standard analysis. The idea is that, since the discrete part of a program is reactive, it computes in an infinitesimal time, while the continuous part is computed in standard time. So, every time there is a zero-crossing, many things can happen, and can be formalized.

4.4.2 Discussion

Such a framework could be useful in the very long term, if ever (see Section 4.4 first).

4.5 Paper [BH01b]

Title	: A clocked denotational semantics for LUCID-SYNCHRONE in COQ
Authors	: Sylvain Boulmé and Grégoire Hamon
Published in	: Long version of a LPAR 2001 paper [BH01a]

Abstract : Synchronous languages [Hal93] have been designed to help in the conception of reactive systems, especially critical reactive systems (planes, power plants control...). Synchrony is a program property which ensures bounded reaction-time and memory at execution. Synchronous languages statically check this property. However, in a critical context, it may be needed to have it formally proved, or more generally to prove program properties.
In this work we are interested in Lucid-Synchrone [PCCH01] (LS for short), a data-flow synchronous language. We present here a natural and shallow embedding of LS into the Coq proof assistant. This embedding concerns both the dynamic and the static semantics of the language, such that synchrony analysis is obtained for free. Moreover, it gives us a denotational semantics of LS in Coq and is thus a good starting point for designing a prover for LS programs in Coq, following [Fil99, Par95] approach. This semantics can also be used to experiment with the language : we have used it here to propose a notion of recursive functions for LS, as a generalization of recursive streams.
The main originality of this work is to apply the "*clocks as types*" paradigm (see [Cas92]) in the design of a formal semantics for a synchronous language. This paradigm consists in expressing static synchronization constraints with a restricted form of dependent types. We show here that such a type system is a subsystem of Coq type system

4.5.1 Synthesis

This work is an expression of the language Lucid Synchrone (LS) in Coq. Since this language manipulates sampled data-flows, they are (naturally) represented by co-inductive streams in Coq, as well as clocks (a stream of bits to define the "instants"). It shows that the constructions of Lucid Synchrone can be embedded in Coq in a *shallow* way, meaning that the semantics is preserved : the computations made by a LS program are faithfully reflected by the computations (β-reduction) of the Coq translation. Therefore, the Coq translation can be thought of a denotational semantics for LS. Even the delay operator (`fby`) is treated.

4.5.2 Discussion

This work is in tune with what is currently being looked at at CRI (see Emilio Gallego's work). One could adapt this work to Faust, so as to express Faust into Coq, giving for free a new semantics. Features like clock inference, equivalence and synchrony can be simplified, even in the multirate case, apparently. As underlined for LS in the paper, one can also take advantage of the result of type, interval and vector-size inferences ultimately performed by the Faust compiler to specify them explicitly in the Coq translation. Indeed, Coq usually needs more information than the source program ; it cannot infer everything since it relies on an extremely powerful and generic type system.

One of the objectives regarding Faust, of course, would be to prove formally, and in Coq, properties of specific Faust programs, like boundedness for signals, no underflow/overflow of buffers if there is some (multirate feature), etc.

4.6 Paper [CMPP08]

Title	: Abstraction of Clocks in Synchronous Data-flow Systems
Authors	: Albert Cohen, Louis Mandel, Florence Plateau and Marc Pouzet
Published in	: The Sixth ASIAN Symposium on Programming Languages and Systems (APLAS)

Abstract : Synchronous data-flow languages such as Lustre manage infinite sequences or streams as basic values. Each stream is associated to a clock which defines the instants where the current value of the stream is present. This clock is a type information and a dedicated type system - the so-called clock-calculus - statically rejects programs which cannot be executed synchronously. In existing synchronous languages, it amounts at asking whether two streams have the same clocks and thus relies on clock equality only. Recent works have shown the interest of introducing some relaxed notion of synchrony, where two streams can be composed as soon as they can be synchronized through the introduction of a finite buffer (as done in the SDF model of Edward Lee). This technically consists in replacing typing by sub-typing. The present paper introduces a simple way to achieve this relaxed model through the use of clock envelopes. These clock envelopes are set of concrete clocks which are not necessarily periodic. This allows to model various features in real-time embedded software such as bounded jitter as found in video-systems, execution time of real-time processes and scheduling resources or the communication through buffers. We present the algebra of clock envelopes and its main theoretical properties.

4.6.1 Synthesis

This paper focuses on extending the core feature of synchronous languages, that is to say *clocks*, and weakening the constraint under which streams, each coming with its own clock, can be composed. As usual, a clock is represented as a stream of instants (booleans, 0/1), and the condition for two clocks to be compatible can be stated in a first approximation as : we want *in average* the same amount of instants to produce a value, all the trick being to define precisely what one means by *in average*. For instance, the affine relation of [BT13] of Section 4.1 is one possible answer.

Here, the answer is a little bit different. It makes use of a subtyping relationship between clocks, of precedence, and allows for *roughly periodic* clocks (that are called aperiodic), that is to say periodic clocks on a large scale but not periodic on the microscopic scale. For instance, a clock that has exactly one 1 value either at instant $3i$, $3i+1$ or $3i+2$ is acceptable, even if the particular instant varies according to i (phase change). It can be plugged to other compatible clocks by a limited use of buffers, provided a check that the source clock always produces 1 before the sink clock needs a value (causality relation).

Other operations, such as clock composition (two clocks w_1 and w_2 can be composed as such : w_1 emits a value, either 0 or 1, at the rate defined by the 1s of w_2), precedence or clock abstraction are defined. Notice that this reminds in some ways of the abstract interpretation framework. This should not be very surprising given the name of the first author.

4.6.2 Discussion

This work seems to go too far w.r.t. the needs of Faust. At least for the moment. there doesn't seem to be much use of aperiodic clocks for audio signals, except, maybe, to handle multiple Faust programs running concurrently on a network with loose but somewhat constrained latency properties.

4.7 Paper [EL03]

Title	: The semantics and execution of a synchronous block-diagram language
Authors	: Stephen A. Edwards and Edward A. Lee
Published in	: Science of Computer Programming

Abstract : We present a new block diagram language for describing synchronous software. It coordinates the execution of synchronous, concurrent software modules, allowing real-time systems to be assembled from precompiled blocks specified in other languages. The semantics we present, based on fixed points, is deterministic even in the presence of instantaneous feedback. The execution policy develops a static schedule–a fixed order in which to execute the blocks that makes the system execution predictable. We present exact and heuristic algorithms for finding schedules that minimize system execution time, and show that good schedules can be found quickly. The scheduling algorithms are applicable to other problems where large systems of equations need to be solved.

4.7.1 Synthesis

The synchronous language discussed here is at a lower level than all the other languages discussed in this review. It handles block-diagrams where every block implements a boolean function. The constraint on the function is that it produces some output even when some input is not determined yet. For example, the boolean disjunction is of this sort : $a \vee 1$ is 1 even if a is undefined and, whatever the value of a is, it will not change ever after. So, the booleans are equipped with a particular lattice structure where the bottom element is $\perp$ (the undefined) and 0 and 1 are incomparable. Thanks to this feature, blocks can be arbitrarily composed, even plugged into themselves (instantaneous feedback) : what one gets is more and more information, but it can never change. Since the quantity of outputs is bounded, this process is finite, deterministic, and reaches a *fixed point* (that technically corresponds to a fixed point in the derived lattice of the n outputs).

The drawback is that blocks must then be evaluated several times : first with all of its inputs undefined ($\perp$), and over and over until no more output becomes defined. The crux is to find a good schedule for the order of the evaluation of blocks, since this can dramatically change the number of passes one must perform (which is at most n passes on all the blocks in a row, if n is the total number of outputs).

This paper makes use of a decomposition of the calculation of a fixed point in a lattice (Bekić's theorem) and of operation research algorithms to find an optimal schedule.

4.7.2 Discussion

One can surely shift from the boolean domain to any set of values, provided one adds a $\perp$ undefined element and forms an elementary lattice, where two different numbers are never comparable. For instance, with natural numbers, it should be possible to have a "+1" operator fed into itself : the only thing it will produce is $\perp$, and not, as we could expect, an ever growing and never stable sequence.

So the feedback here has not much to do with the usual "delay" feedback of Faust or of other synchronous programming languages. Recursivity is also not implementable in this way, due to the monotonous and static nature of the functions considered here : outputs are fixed once and for all (and may be never fixed). Having such a primitive in Faust would greatly complicate the analysis and the typing of correct programs.

Since there is only delayed feedback in Faust, the evaluation order is much simpler, and does not need such optimization techniques.

4.8 Paper [MPP11]

Title	: Static scheduling of latency insensitive designs with Lucy-n
Authors	: Louis Mandel, Florence Plateau and Marc Pouzet
Published in	: FMCAD 2011

Abstract : Synchronous functional languages such as Lustre or Lucid Synchrone define a restricted class of Kahn Process Networks which can be executed with no buffer. Every expression is associated to a clock indicating the instants when a value is present. A dedicated type system, the clock calculus, checks that the actual clock of a stream equals its expected clock and thus does not need to be buffered. The n-synchrony relaxes synchrony by allowing the communication through bounded buffers whose size is computed at compile-time. It is obtained by extending the clock calculus with a subtyping rule which defines buffering points.

This paper presents the first implementation of the n-synchronous model inside a Lustre-like language called Lucy-n. The language extends Lustre with an explicit buffer construct whose size is automatically computed during the clock calculus. This clock calculus is defined as an inference type system and is parametrized by the clock language and the algorithm used to solve subtyping constraints. We detail here one algorithm based on the abstraction of clocks, an idea originally introduced in [5]. The paper presents a simpler, yet more precise, clock abstraction for which the main algebraic properties have been proved in Coq. Finally, we illustrate the language on various examples including a video application.

4.8.1 Synthesis

This paper is about an n-synchronous language called Lucy-n and shows some potential applications of it for video signal processing. This language is synchronous, and allows to consider several clocked processus, with different clocks. As usual, clocks are represented as infinite repetitive streams of `0` and `1`, in a very regular expression fashion, simplified of course. What is allowed is clocks of the form `prefix.(period)*`, called ultimately periodic clocks in [BT13].

The paper studies the conditions under which they can be composed without any underflow or overflow, potentially with the introduction of (bounded) buffers, through a *clock subtyping* relation. To compute this relation, constraint are generated. Since it is computationally infeasible to solve them exactly (there is no hint about the complexity of the problem, however), an abstraction is proposed, which is an improvement over the paper [CMPP08]. The proposed abstraction is a clock *envelope*, composed of two straight affine lines that bound the (integral of the) clock when represented as a discrete function over time.

4.8.2 Discussion

This paper is surely useful when it comes to speaking about multirate Faust. In a sense, Faust is simpler, since clocks are pure periodic clocks, and have always the shape `(000...01)*`. It is mainly useful at the late stages of the compilation process, when the question of defining buffer sizes arises. It remains to understand if all this machinery is needed, or only a simplification of it (the generated constrains, in the Faust case, could be very simple), or even nothing of it. The buffer computation algorithm can be useful if one, for instance, wants to consider multiprocessor execution of a single Faust program.

4.9 Paper [RRJ+14]

Title	: Automating formal proofs for reactive systems
Authors	: Daniel Ricketts, Valentin Robert, Dongseok Jang, Zachary Tatlock and Sorin Lerner
Published in	: PLDI 2014

Abstract : Implementing systems in proof assistants like Coq and proving their correctness in full formal detail has consistently demonstrated promise for making extremely strong guarantees about critical software, ranging from compilers and operating systems to databases and web browsers. Unfortunately, these verifications demand such heroic manual proof effort, even for a single system, that the approach has not been widely adopted.
We demonstrate a technique to eliminate the manual proof burden for verifying many properties within an entire class of applications, in our case reactive systems, while only expending effort comparable to the manual verification of a single system. A crucial insight of our approach is simultaneously designing both (1) a domain-specific language (DSL) for expressing reactive systems and their correctness properties and (2) proof automation which exploits the constrained language of both programs and properties to enable fully automatic, pushbutton verification. We apply this insight in a deeply embedded Coq DSL, dubbed REFLEX, and illustrate REFLEX's expressiveness by implementing and automatically verifying realistic systems including a modern web browser, an SSH server, and a web server. Using REFLEX radically reduced the proof burden : in previous, similar versions of our benchmarks written in Coq by experts, proofs accounted for over 80% of the code base ; our versions require no manual proofs.

4.9.1 Synthesis

This very interesting paper describes how, for a toy reactive language called REFLEX, one can express properties that are verified in a fully automated way in Coq. Some caveats are needed though :

— this requires to have a hand-programmed library of basic facts in Coq ;

— properties are expressed w.r.t. to traces, which are an abstraction over programs ;

— both the REFLEX and the property languages are toy languages, so not able to express everything ;

— a proof-script generator, that takes a program and its properties and generates Coq's proof scripts (with the use of tactics), is also needed.

The REFLEX language itself is not embedded into Coq, only the abstraction (the traces it generates) is. They first implemented the interpreter through a use of the Ynot library, where the properties/axiomatization of primitives are expressed as pre- and post-conditions that the axiomatization verifies over traces. Those assertions are verified by hand at a low level. Then, they define in Coq a behavioral abstraction function that extracts traces from programs. The programs are in the form of ASTs, decorated with type information. The adequacy of the abstraction w.r.t. the interpreter is verified. The last piece is a small language to express properties on the now available traces, for instance "temporal" properties over traces, such as "immediatelyBefore", or non-interference. This last property is the most involved one, since the execution of the language is non-deterministic. The automation of the proofs that all the traces generated by the behavioral abstraction of the program respect the stated property is then performed by tactics that inductively build a proof. Once again, the case of non-interference is the most involved one. Tactics are nevertheless incomplete.

4.9.2 Discussion

The approach presented here shows something we really should put forward for Faust too : a small language of properties that one can express, for instance, as annotations, and an automatic tool that, if it can, generates a proof script that is, hopefully, successfully verified by Coq. This may be combined with a definition of the semantics of Faust in Coq or some even simpler abstractions, for instance intervals.

Another thing one can retain from this research work is the use of the Ynot library to express non-terminating, imperative programs that are a collection of handlers to react to events. Here, it is used to implement an interpreter. Unfortunately this seems quite far from Faust : the handlers of Faust are very simple (`+`, `delay`, ...) and there are many of them, while the handlers considered in this paper are more complex things (typically web servers, or models of them). Maybe this would be a good approach for a subset of the Faust's larger input language, not of Core Faust.

4.10 Paper [BJMP13]

Title	: A Synchronous Embedding of Antescofo, a Domain-Specific Language for Interactive Mixed Music
Authors	: Guillaume Baudart, Florent Jacquemard, Louis Mandel and Marc Pouzet
Published in	: EMSOFT 2013

Abstract : Antescofo is recently developed software for musical score following and mixed music : it automatically, and in real- time, synchronizes electronic instruments with a musician playing on a classical instrument. Therefore, it faces some of the same major challenges as embedded systems. The system provides a programming language used by composers to specify musical pieces that mix interacting electronic and classical instruments. This language is developed with and for musicians and it continues to evolve according to their needs. Yet its semantics has only recently been formally defined. This paper presents a synchronous semantics for the core language of Antescofo and an alternative implementation based on an embedding inside an existing synchronous language, namely ReactiveML. The semantics reduces to a few rules, is mathematically precise and leads to an interpreter of only a few hundred lines. The efficiency of this interpreter compares well with that of the actual implementation : on all musical pieces we have tested, response times have been less than the reaction time of the human ear. Moreover, this embedding permitted the prototyping of several new programming constructs, some of which are described in this paper.

4.10.1 Synthesis

Antescofo is a DSL for music that serves a very different purpose than Faust : it is used to have both digital and analog (humans beings) performers. So, Antescofo must be able to follow a score, estimate the tempo, and play according to events that are the notes played by the human musicians. Moreover, humans can play wrong notes, and the event can be not the one expected. Antescofo integrates such mechanisms, so as to be as close as possible to human players.

This gives a perfect match with synchronous languages : they react to external events. This is why expressing the semantics of Antescofo and programming an interpreter for it in a language such as Reactive ML make much sense and, at the same time, are not too difficult tasks.

4.10.2 Discussion

Given the totally different nature of Antescofo and Faust, it is not likely that one can reuse this work as a basis for a work on Faust.

5 Conclusion

This paper presents an overview of ten significant research works related to synchronous programming, possibly with some connection with computer music. We provided a rather detailed presentation of each of these papers, and summarized our findings regarding their possible impact on the Faust audio programming language and its ecosystem.

6 Acknowledgements

The author is indebted to Pierre Jouvelot, Yann Orlarey, Imré Frotier de la Messelière and Emilio Gallego Arias for the many discussions, some pointers to articles about this topic and proofreading of this paper.

Références

[BBCP12] Albert Benveniste, Timothy Bourke, Benoit Caillaud, and Marc Pouzet. Non-Standard Semantics of Hybrid Systems Modelers. *Journal of Computer and System Sciences (JCSS)*, 78(3) :877–910, May 2012. Special issue in honor of Amir Pnueli.

[BH01a] Sylvain Boulmé and Grégoire Hamon. Certifying synchrony for free. In Robert Nieuwenhuis and Andrei Voronkov, editors, *LPAR*, volume 2250 of *Lecture Notes in Computer Science*, pages 495–506. Springer, 2001.

[BH01b] Sylvain Boulmé and Grégoire Hamon. A clocked denotational semantics for lucid-synchrone in COQ. manuscript, 2001.

[BJMP13] Guillaume Baudart, Florent Jacquemard, Louis Mandel, and Marc Pouzet. A synchronous embedding of Antescofo, a domain-specific language for interactive mixed music. In *Thirteen International Conference on Embedded Software (EMSOFT'13)*, Montreal, Canada, September 2013.

[BP13] Timothy Bourke and Marc Pouzet. Zélus, a Synchronous Language with ODEs. In *International Conference on Hybrid Systems : Computation and Control (HSCC 2013)*, Philadelphia, USA, April 8–11 2013. ACM.

[BT13] Adnan Bouakaz and Jean-Pierre Talpin. Design of safety-critical java level 1 applications using affine abstract clocks. In *Proceedings of the 16th International Workshop on Software and Com-*

pilers for Embedded Systems, M-SCOPES '13, pages 58–67, New York, NY, USA, 2013. ACM.

[CMPP08] Albert Cohen, Louis Mandel, Florence Plateau, and Marc Pouzet. Abstraction of Clocks in Synchronous Data-flow Systems. In *The Sixth ASIAN Symposium on Programming Languages and Systems (APLAS)*, Bangalore, India, December 2008.

[DGP08] Gwenael Delaval, Alain Girault, and Marc Pouzet. A Type System for the Automatic Distribution of Higher-order Synchronous Dataflow Programs. In *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Tucson, Arizona, June 2008.

[EL03] Stephen A. Edwards and Edward A. Lee. The semantics and execution of a synchronous block-diagram language. *Sci. Comput. Program.*, 48(1) :21–42, 2003.

[MPP11] Louis Mandel, Florence Plateau, and Marc Pouzet. Static scheduling of latency insensitive designs with lucy-n. In Per Bjesse and Anna Slobodová, editors, *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, pages 171–175. FMCAD Inc., 2011.

[PEB07] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. *Compiling Esterel.* Springer, 2007.

[RRJ$^+$14] Daniel Ricketts, Valentin Robert, Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Automating formal proofs for reactive systems. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 47. ACM, 2014.