In Proceedings of the 11th International Conference on Systems Engineering (ICSE'96), Las Vegas, Nevada, July 9–11, 1996.

On the desynchronization of synchronous applications

Pascal Aubry and Paul Le Guernic
IRISA/INRIA, Campus de Beaulieu, F35042 Rennes Cedex

Abstract

Synchronous data-flow programming is based on the abstraction of the continuous time into a discrete one. It offers verification and proof techniques appreciated by programmers of critical reactive systems. Armed with a strong semantics, the synchronous data-flow languages (SIGNAL, LUSTRE,...) generally lead to strongly synchronized executions. In the case of distributed implementations, some constraints, introduced by the specification itself, are very prejudiciable in terms of performance.
The desynchronization of synchronous applications is a paradox in itself. To conciliate rigorous specification and efficient implementations, we briefly present a new model based on a minimal semantics. Thanks to this model we show the desynchronization process leading to desynchronized executions from a synchronized specification.

1 Relaxing from synchrony

Based on the hypothesis of a discrete logical time, the synchronous languages [1, 2] (SIGNAL [3, 4], LUSTRE [5], ESTEREL [6]) have proved their efficiency for the design of critical and safe real-time applications. They are characterized by strong semantics that allows the programmer to use verification and proof techniques. As this is not the purpose of this paper, no need to go on commenting the benefits of the synchronous approach for the programming of reactive systems [1].

At the opposite, devil's advocates would tell you on the drawbacks of such a strong semantics. For instance, let us consider the mathematical sequence:

$$\forall n \in I\!N,\ n \geq 2 \Longrightarrow A_n = g(A_{n-1}, f(A_{n-2})).$$

Of course, no need to raise the synchronous approach to program such a trivial sequence, but, never mind, it can be written in SIGNAL this way:

```
(| a := g( a $ 1,b ) | b := f( a $ 2 ) |) /b
```

As SIGNAL is declarative, the source program is a system of equations, where `a` and `b` are signals, i.e. unbounded sequences of values, `$ n`, where `n` is an integer, symbolyses a `n`-delay. `a` and `b` are, because of the equations, "*synchronous*": they are present at the same instants. At each instant, the value of `a` depends on the values of `a$1` and `b`, while the value of `b` depends on the value of `a$2`. Finally, the successive occurences of the signals `a` and `b` are temporally ordered, and "`/b`" means that the scope of `b` is reduced to the process. The behaviour of this program, implemented with actual synchronous compilers, can be represented by the diagram on figure 1, showing the occurrences of the flows `a` and `b`, temporal dependencies ($\rightarrow$), data

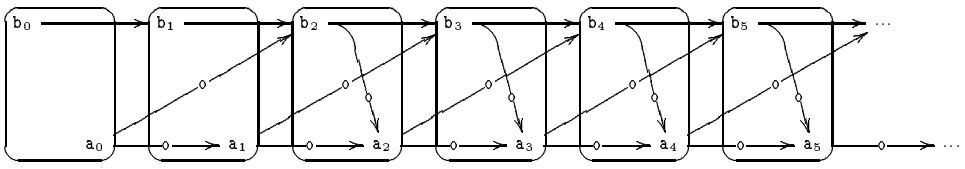

Figure 1: A synchronous specification.

dependencies ($\rightarrowtail$) and synchronizations between flows (framed sets).

People unused to the synchronous approach may think "What an intricate thing to specify such a simple application!", but that is the price paid for the benefits of the synchronous approach [1]. Strong semantics

yields over-constrained specifications for people who only want to simulate. But many constraints (relations) in this behaviour have been introduced by the specification itself, and are not useful for the simulation of the program. For instance, the synchronization between `a` and `b` is the consequence of the extension of SIGNAL functions to sequences of values. As `b` is a temporary flow, not part of the interface of the process producing `a`, `a` and `b` could be desynchronized without changing the sequence of values of `a`. For the same reasons, even some temporal dependencies between the successive occurrences of `b` can be erased. By splitting odd and even occurrences of `b` in two (two new signals), we get a new behaviour respecting the original specification, from an external point of view (see on figure 2). Indeed, as an external observer only sees the occurrences

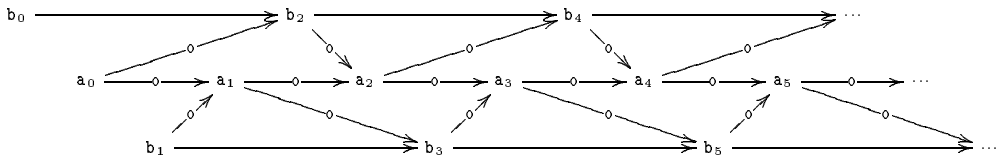

Figure 2: A possible observation of the previous program.

of `a`, if they are still correctly ordered, he cannot guess the trick. At the very most, he will perhaps suspect something when discovering that his program can run 2 times faster than usually. Indeed, this behaviour, implemented on a multi-processor architecture, obviously has a much better computation frequency than the first one: the speed-up can reach 2 if neglecting communication durations.

Especially in the case of distributed applications, performance requirements may make the programmer turn towards asynchronous languages (ADA [7], OCCAM [8], CSP [9],...) or mixed synchronous/asynchronous languages (CRP [10]), losing then 1) the global determism of the application and 2) some proof and verification techniques so precious with critical applications. How can we reach deterministic and efficient distributed implementations, while preserving all the benefits of the synchronous approach?

The mechanisms involved in the transformations above seem simple but become quite complex as soon as we want to formalize them. Moreover, the automation of such techniques requires a fine knowledge of dynamic behaviours to appreciate the consequences of different transformations on the original specification; especially, is the original semantics still respected?

In this paper we investigate a new approach: instead of desynchronizing over-constrained specifications, we give SDF languages a minimal semantics: data dependencies, explicit serializations and synchronizations. In section 2 we describe the compilation process from specification to final implementation. We also explain the properties users of SDF applications want to verify at run-time. Section 3 gives an overview of the model we use to characterize these properties and presents different possible implementations of SDF languages. It also gives results obtained on the common transformations performed to get get distributed implementations: data communications, duplication of computations, retiming, reinforcement of serializations.

2 Designing synchronous data-flow applications

In this part we give an overview of the design process of an application with a data-flow synchronous language and show some properties users may want compilers to be able to guaranty. We conclude on the need of a new model for synchronous data-flow applications.

2.1 Inferring synchronous data-flow languages

Primary compilation. By primary, please do not understand trivial: the compiler can not only be a machine transforming input source into executable code. It is a formal system able to reason logically. Of course, as many compilers, its first job is to parse the input and perform some classical operations such as type-checking; but its final goal is to produce what we call the "*system description*" thanks to the semantics of the language (figure 3). Nothing is assumed on this description but let us remark it is always an instantaneous description[1] in the classical synchronous data-flow languages already implemented. Lustre is compiled into automata and SIGNAL into a specific representation called "*Synchronized Data-Flow Graph*". The constraints

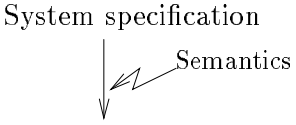

Figure 3: Primary compilation.

[1] i.e. describing the reaction of the system to its environment in a logical instant.

of the description is made up of the explicit constraints of the programmer's specification and of the implicits constraints induced by the semantics of the language: dependencies and synchronizations between occurences.

Distribution, reinforcement. At first, the system description is distributed in a way we will not precise here. Distribution directives can be given by the programmer at source-level (functionnal distribution), directly expressed on the system description (quantitative distribution) and may be incomplete. In this case, a strategy is needed to completly map the description onto a finite set of processors. Because of the cost of dynamic schedulers at run-time, a procedurization of the program is performed to schedule statically as much as possible. Indeed, the overhead of dynamic schedulers may induce response times out of real-time requirements. A reinforcement of the dependencies is performed and leads, after some possible optimizations, to a new representation of the original program called "*system implementation*" (see on figure 4). This interpretation of the design process is deliberately simplified to make it easier to be understood. In particular, optimizations can be performed all along the transformations of the initial description, and back-tracking algorithms leading to more and more efficient implemantations can be used. An implementation is made up of dependencies between occurences.

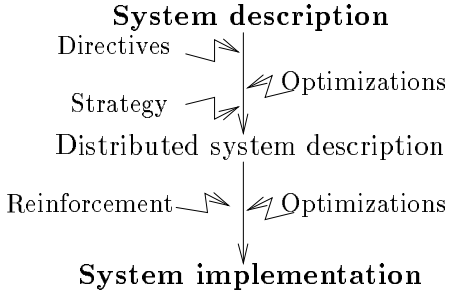

Figure 4: Distribution, reinforcement.

Execution. The system implementation is immersed in its environment and the execution can be observed by a (possibly virtual) observer. An observation of an implementation is thereafter the result of an execution, and is made up of a set of dated occurences, that can be ordered on each processor.

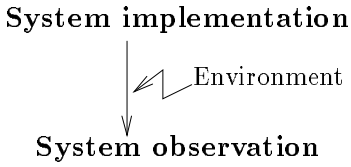

Figure 5: Execution.

2.2 The user's point of view

A programmer designing real-time critical systems chooses (data-flow or imperative) synchronous languages at first to benefit from a large panel of proof and verification techniques. He (she) also wants some efficient implementations, preserving all the constraints of its specification or only parts of them: we have seen in part 1 that desynchronizing and relaxing dependencies can lead to much better implementations. The user may want those kind of transformations (optimizations) but needs to characterize them. More precisely, he wants to know in which way an implementation provided by the compiler corresponds to its specification. Furthermore, a user-friendly programming environment should be able to provide different implementations depending on the user's will. Finally, the implementations should be able to answer the following questions: 1) Is their response time bounded? Is it enough bounded[2] to satisfy real-time constraints of the environment? 2) Is the memory needed bounded? 3) Is it fair or will the user see some parts of its application run infinitely faster than other ones?

2.3 Characterizing implementations with regard to specifications

In the sequel of this paper, we assume that the production of the system description (see above) is extracted from the specification in a single way, only depending on the semantics of the language and the implementation of the compiler itself. We also assume that the description produced is correct, e.g. corresponding to the user's specification and implementable (deterministic for us). The problem is then: given a specification, how can we insure that all the possible observations of an implementation will respect some properties of the description? At first let us enumerate the most common of these properties:

Data-Flow equivalence. We want the initial dependencies of the description to be respected by any possible observation of the implementation. This way sequences of values observed are exactly the ones specified by the user.

Synchronous Data-Flow equivalence. Those kind of implementations should preserve not only sequences of values but also the logical instants of the description. In particular, no overlapping between instants should be allowed. As no global notion of time exists in the case of multi-processor implementations, this equivalence can be global or only local (applying to occurences on the same processor).

[2] In our point of view, timing requirements are verified *a posteriori*. Quantitative criteria are not treated here.

External DF and SDF equivalences. Those kind of implementations should respectively be Data-Flow and Synchronous Data-Flow equivalent but only with regard to interface flows of the specification (input and output flows).

2.4 Requirements for a new model

As briefly said before, descriptions used by synchronous data-flow languages are instantaneous: they describe the reaction of the application to its environment during one logical instant. In particular, implicit dependencies from any occurence of an instant n to any occurence of the instant $n+1$ are added in the implementations provided by actual compilers. This way, if the implementations respect the original dependencies in each instant, the synchronous data-flow equivalence is easily guarantied.

The goal of this paper is to provide a new model able to describe complex transformations, such as the ones shown in the introduction: distribution of labels (common approach of distribution), and distribution of occurences (for instance spliting odd and even occurences of a label onto different processors). The model should also describe the last design phase, e.g. the reinforcement of the initial dependencies leading to: totally **asynchronous executions**, **synchronized executions** where the mapping of logical instants onto the physical time tolerates no overlappings, **k-desynchronized executions** where overlappings of logical instants are allowed but bounded. Of course, since desynchronizations and impoverishment of dependencies may take place between more than one logical instant, any instantaneous description is unsufficient. Finally, the model used should be able to characterize some extended properties such as memory-bounding, response time-bounding and fairness.

3 A new model for Synchronous Data-Flow Applications

As we wanted our model to be able to describe applications all along the design process, we firstly introduced *descriptions*, consisting in a set of constraints (dependencies and synchronisations) between the occurences of the signals of the program. Dependencies are transformed and reinforced into *implementations*, and implementations are observed at run-time to give *observations*. The use of a single formalism from the specification until the simulation is set possible by the different meanings of dependencies: when $\mathtt{x} \rightarrow \mathtt{y}$ means that `y` should preceeded by `x` (constraint) in descriptions, it means that `x` will always be present before `y` (scheduling strategy) in implementations and that `x` has been observed before `y` in observations. This allows us to extract some properties of applications and show that some of them can be preserved by the different transformations.

A minimal semantics The classical justification of the synchronous approach says that it is reasonable to consider a reaction as instantaneous if the time needed to react is lower than what the environment can impose. In other words, the system must consume inputs, produce corresponding outputs and then get ready for the next instant before new inputs arrived. This interpretation is no more sufficient for desynchronized implementations. A minimal semantics of SDF languages is considered in the sense that only data dependencies and user serializations are taken into account; in particular, implicit serializations between occurences of distinct instants do not exist any more. When we had to break serializations with the old classical point of view, we only need now to add serializations to get implementations constrained enough to verify properties.

No hypothesis is made on the implementation finally chosen at specification level. Synchrony is used to get a strong semantics indispensable for the appliance of formal tools, but constraints do not affect implementations any more.

Flexibility versus realism. The design of the model continously knocked against the following paradox: it must be flexible enough to fit with a large panel of transformations and rigid enough to allow the extraction of properties. The goal is also to give a consistent operational semantics to data-flow synchronous languages: SIGNAL in particular, but LUSTRE could also be represented by the model, then providing an extension of the asynchronous semantics based on simple partial orders [11]. Thus the constraints that descriptions should observe are minimal to get a realistic model where classical transformations can be easily described.

Simultaneousness, serializations and data dependencies: a complete mixture. An occurrence is the temporal existence of a value belonging to a domain $\mathcal{D}$. Occurrences are labelled. As type control is for us a technical part made at early compile-time, no hypothesis is made on the domain $\mathcal{D}$. Pre-executions are sets of occurrences armed with three relations: simultaneousness, serializations and data dependencies. These relations are constrained to provide realistic graphs.

No need to argue on the need of serializations ($\rightarrow$): they are commonly used in graph theory. Completed with the simultaneousness, the model provides a good generalization of labelled partial orders [11, 2]. It may seem too general because allowing non deterministic executions (rejected in SIGNAL), not memory-bounded or non fair executions (also rejected), but these issues can all be characterized. The simultaneousness ($\odot$) is directly stemmed from the synchronous approach: it explains the adherence of some occurrences to the same logical instant. Projected on the input/output occurrences, serializations and simultaneousness give what is called the observational semantics of an application. Finally completed by (direct) data dependencies ($\rightarrowtail$), the model is able to represent real-time applications since their description until their final implementation, which makes easier the proofs of the algorithms involved in the compilation process.

As ten more pages would just contain the definitions and theorems used to characterize the transformations performed on descriptions, we prefer to give results proved on the most common tranformations.

3.1 Final Implementations

Hypothesis on descriptions. One can consider that LUSTRE programs are top-down built, by successive refinements of the main clock of the program. At the opposite, SIGNAL programs can be built bottom-up by composing abstracted *processes* used as "black boxes". As the composition may result in new clocks, the description produced is in general case a forest of clock-hierarchies [12]. To insure deterministic implementations, we say that SIGNAL descriptions are implementable when the forest of clock-hierarchies is reduced to a tree; this way a single main clock of the program rules the execution. In all cases, we know that a flow is present at least as often as all the other flows, and that the occurences of this flow are totally ordered.

Total desynchronization. Thanks to the minimal semantics, totally desynchronized implementations are achieved by doing nothing! Dynamic scheduling is made at run-time. The problem is that this leads generally to $\mathcal{M}$-unbounded, $\mathcal{RT}$-unbounded and not fair implementations, not really interesting for users. Asynchronous implementations are not provided.

Strong synchronization. At the opposite, strong synchronization is achieved by the serialization of any computation of an instant n with any computation of the instant $n+1$ of the main clock of the program. This way overlapping between logical instants is impossible. We prove that this insures $\mathcal{M}$-bounding, $\mathcal{RT}$-bounding and fairness.

This is done in particular for the generation of mono-processor sequential code [13]: once instants are serialized, dependencies "in the instant" are totally reinforced to schedule the program statically. This provides to users SDF-equivalent implementations.

This is also done for multi-processor implementations [14] but dependencies "in the instant" are reinforced only sometimes and between computations located on the same processor. Possible parallelizations left free are solved at run-time by a reduced dynamic scheduler. Depending or the duplication or not of some flows (especially the "control" part of the program), the user get SDF or at least eSDF-equivalent implementations.

k-desynchronization. Distributed implementations described above only benefit from parallelism "in the instant" because of the resynchronizations of all the processors (sometimes thanks to void messages) at the beginning of each instant. As potential parallelism also exists between instants, we provide users k desynchronized ($k > 0$) implementations: any computation of an instant n is serialized with any computation of the instant $n+k$ of the main clock of the program. Thanks to the model, we prove that those kind of implementations are $\mathcal{M}$-bounded, $\mathcal{RT}$-bounded and fair.

3.2 Common transformations of descriptions

We describe here the common operations performed on descriptions by compilers: communicating data, duplicating computations, retiming, splitting labels. We give for each transformation the properties preserved by implementations.

Communicating data In the distribution stage, communication nodes are added on the instantaneous graph to spread data wherever they are needed. The effects of the communication of the occurences of any signal **x** between processors preserves SDF-equivalence.

Duplicating computations For qualitative (fault tolerant duplications) or quantitative (rate increasement) motivations, computations can be duplicated on different processors. In general, some serializations are lost during the transformation. Nevertheless, we prove that eDF and eSDF-equivalences are preserved when the computations of external (input/output) signals are not duplicated.

Retiming The principle of retiming is to delay or bring forward computations. It is used in particular to reduce the number of state variables in programs and make easier complex operations such as model-checking. As a short example, we see the transformation of a simple system of equations changing the set of variables from $\{\mathtt{a},\mathtt{b}\}$ to $\{\mathtt{x}\}$: $\left\{\begin{array}{l}\mathtt{x}_n = \mathtt{F}(\mathtt{a}_{n-k},\mathtt{b}_{n-k})\\ \mathtt{y}_n = \mathtt{G}(\mathtt{x}_n)\end{array}\right. \Longrightarrow \left\{\begin{array}{l}\mathtt{x}_n = \mathtt{F}(\mathtt{a}_n,\mathtt{b}_n)\\ \mathtt{y}_n = \mathtt{G}(\mathtt{x}_{n-k})\end{array}\right.$. By affecting to variables weights equal to the amount of memory needed to store them, retiming can also be used to reduce memory allocation[3]. These changes are explained on an instantaneous description by translations of computations. Neither occurences nor dependencies are changed when retiming. This may only result in the loss of eSDF-equivalence if $\mathtt{x}$ is an external signal and SDF-equivalence in all cases.

Splitting labels Essentially for quantitative motivations, programmers need to split successive computations of a signal on many processors, depending on a *modulo* counter for instance. Especially used when designing "regular" applications (see our example in section 1), this feature helps finding quick and good distributions and should therefore be automized. This transformation can be explained thanks to communications of data and duplications of computations. eSDF-equivalence is thus proved when dealing with internal signals.

Perspectives

Thanks to a model based on a minimal semantics, we conciliates the rigour of synchronous specifications and the efficiency of desynchronized implementations. It brings a new way to handle the relaxation of synchrony and dependencies of over-constrained specifications. At the moment, only strongly synchronized distributions of SIGNAL have been implemented but we believe that the k-desynchronization will open up new horizons that are currently investigated.

References

[1] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

[2] Albert Benveniste, Paul Caspi, Paul Le Guernic, and Nicolas Halbwachs. Data-Flow Synchronous Languages. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Lecture Notes in Computer Science 803, Proc. of the REX School/Symposium, Noordwijkerhout, Netherlands*, number 803, pages 1–45. Springer–Verlag, June 1993.

[3] Paul Le Guernic and Thierry Gautier. Data-flow to von neumann: the SIGNAL approach. In J.L. Gaudiot and L. Bic, editors, *Advanced topics in data-flow computing*, pages 413–438. Prentice Hall, 1991.

[4] Paul Le Guernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, septembre 1991.

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

[6] F. Boussinot and R. De Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

[7] N. Gehani. ADA *Concurrent Programming*. Prentice-Hall, 1984.

[8] David May. Communicating processes and OCCAM. INMOS *Technical Note 20*, 1987.

[9] Charles Antony Richard Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[10] G. Berry, S. Ramesh, and R.K. Shyamasundar. Communicating Reactive Processes. In *20th ACM Symposium on Principles of Programming Languages*, pages 85–98, Charleston, South Carolina, 1993.

[11] Alain Girault. *Sur la répartition de programmes synchrones*. PhD thesis, Institut National Polytechnique de Grenoble, January 1994. in french.

[12] T. Pascalin Amagbegnon, Loïc Besnard, and Paul Le Guernic. Implementation of the data-flow synchronous language SIGNAL. In *Programming Languages Design and Implementation*, pages 163–173. ACM, ACM, 1995.

[13] O. Maffeïs. *Ordonnancements de graphes de flots synchrones; Application à* SIGNAL. PhD thesis, Université de Rennes 1, France, January 1993. in french.

[14] P. Aubry, P. Le Guernic, and S. Machard. Synchronous distribution of SIGNAL programs. In *29th Hawaii International Conference on System Sciences*, volume 1, pages 656–665. IEEE Computer Society Press, January 1996.

[3] needed when inferring fault tolerant implementations by storing the state of the system at each logical instant.