ELSEVIER
Theoretical Computer Science 281 (2002) 31–36
Theoretical Computer Science
www.elsevier.com/locate/tcs

Nivat's processes and their synchronization

André Arnold
Universite de Bordeaux I, LABRI, CNRS UMR 5800, 351 cours de la Liberation, F-33405 Talence, France

Abstract

This short paper retraces how the notion of synchronization of processes introduced by Maurice Nivat in 1979 has evolved over more than 20 years. © 2002 Elsevier Science B.V. All rights reserved.

At the end of the 1970s, Maurice Nivat took interest in the semantics of parallel and concurrent systems. With G. Ruggiu of the *Laboratoire central de recherches* of the French company *Thomson-CSF*, he organized a joint seminar on this topic. That is why his seminal paper on the synchronization of processes [14] appeared in a technical journal of this company and, therefore, was unfortunately not widely known. An extended version of this paper [15] was published in a yet more confidential place.

However, the ideas expressed in these papers have made their way; for instance, they are implemented in the model-checker MEC [4] and are the basis of the AltaRica formalism [6].

In this paper, we wish to explain how these ideas have evolved along time and to compare them with some other similar ideas.

In [14], a process is defined as a set of infinite sequences of actions or events. Each action or event is denoted by a letter from an alphabet A, so that a process is defined as a set of infinite words over A. It appeared very soon that this definition was not general enough, in particular, because it did not include the notion of a deadlocking process, which was nevertheless considered as a very important notion in the last part of the paper.

Therefore, in [15], a process is defined as a triple of languages over A: the set of infinite sequences, the set of finite *terminated* sequences, and the set of all finite sequences the process is able to perform. Clearly the latter set contains all the *prefixes* (or initial subsequences) of the former two.

Such a process is said to be *deadlock-free* (or *nonblocking*) if each finite sequence which is not terminated can be extended by at least one letter into another finite

0304-3975/02/$ - see front matter
PII: S0304-3975(02)00006-3

sequence. It follows that every finite sequence which is not terminated can be extended into a terminated sequence or into an infinite word over A all of whose prefixes are in the process. But it is not guaranteed that such an infinite word belongs to the process. If it is always the case, the process is said to be *safe*.

The above definitions are intuitively quite natural and it is not a surprise that they appear elsewhere, for instance, in the fundamental papers by Ramadge and Wonham on the control of discrete event systems [16,17]. They define a discrete event system as a Nivat's process without infinite sequences, with the restriction that both languages (of all finite sequences and of all terminated ones) are recognizable, and they say that it is nonblocking if every non terminating sequence can be extended into a terminated one, which is exactly the notion of safeness for a process which has no infinite sequences.

The restriction to recognizable processes is quite natural as soon as one is interested in decision problem (whether a process is safe or not) or effective constructions (e.g., of the greatest safe subprocess of a given process) and was considered in [15]. The paper [8] investigates such problems for some kinds of algebraic (or context-free) processes.

The main idea of [14] is to define the interactions between processes (communications, synchronizations) on a very high level of abstraction, so that almost all mechanisms encountered so far in the literature become particular cases of a very general concept: the synchronization vectors.

Actually, the effect of a synchronization mechanism is to force some actions of some processes to be performed simultaneously, or, on the contrary, to forbid their simultaneous occurrences. Therefore, it can be described by giving the list of tuples of action that must or may occur simultaneously (or its complement, the set of tuples that can never occur simultaneously).

Formally, for $i=1,\ldots,n$, let L_i be a process over the alphabet A_i. Let A_i^0 be the set $A_i \cup \{\varepsilon\}$, where ε is a symbol to be interpreted as "no action". A *synchronization vector* is any element of the product $\mathbf{A}=A_1^0 \times \cdots \times A_n^0$. A *synchronization constraint* is a set $I \subseteq \mathbf{A}$ of synchronization vectors

The *synchronized product* of $L_1,\ldots,L_n$ with respect to I is the process over I defined as follows. Its finite (terminated) sequences are those whose projections are finite (terminated) sequences of the corresponding component, its infinite sequences are those whose projections are infinite or terminated.

It is not difficult to convince oneself that communication mechanisms of CSP [10], CCS [13], COSY [12] can be described in this way. Even Petri nets can be seen as synchronized products where the processes are places and synchronization vectors are transitions.

Indeed, the synchronization mechanism proposed in [15] is a little more general: the synchronization constraint is a *process* over I, acting as a controller or supervisor, and the *synchronized product* is defined by the additional condition that its sequences must also belong to the control process. The former definition is a particular case of the latter when the control process consists of all finite and infinite sequences over I. The difference between them is quite similar to the difference between vector addition systems and vector addition systems with states.

However, in a few lines argument, we explained in [15] that a specific control process is not really needed and that the first definition is quite sufficient, because one can apply it to a vector of processes containing also this control process as an additional component and then hide the actions of this control process by taking a homomorphic image of the synchronized product. Actually, the notion of a control process no longer appeared in subsequent papers on synchronized products [2,3] and felt into oblivion. This forgetting was an unforgivable blunder, we completely missed the point that a controller for a system of interacting processes is not a process similar to others as it was established by Ramadge and Wonham [16,17] and, however strange it may appear, the notion of a specific control process was reintroduced in AltaRica [6] without any conscious reminding of this first attempt.

Recognizable processes are those each of the three languages of which is recognizable. Therefore, they can be represented by a finite automaton (or labeled transition system), and the synchronized product of recognizable processes is easily represented by the automaton obtained as a kind of product of the automata representing the component processes. We privately used this construction to build up examples, and even to prove some basic results, until it appeared that the synchronized product of labeled transition systems was indeed "the" basic notion. This is why, in [2], the synchronized product is defined on transition systems, and no longer on processes, although transition systems are just seen as a way of specifying Nivat's processes. Actually, it is conceptually easier to work with transition systems than with languages. As an example, Bergeron [7] drastically simplified some proofs of Ramadge and Wonham just by considering that the supervisor of a "plant" is a transition system rather than a language.

The next step, that still took some time, was to make explicit the fact that a transition system allows us to define not only a triple but a whole bunch of languages, according to the acceptance condition that is chosen for each of them. Then, the process is indeed the transition system, and the triples that actually started the story definitively went out of the picture. It only remained the notion of synchronized products of transition systems [3].

However, the fact that we arrived at this notion as it is explained had an important consequence: the states play no role at all in the synchronization mechanism (in contrast, for instance, with the *s/r model* of Kurshan [11]) and only the actions or events of each component can be used to synchronize them. Indeed, the states of a transition system are inside a black box, that the designer is allowed to open, but the other components of the system to be synchronized are not. It follows that the synchronization product is congruent with respect to the bisimulation equivalence.

The emergence of the synchronized product of transition systems as a basic notion followed many experiments by hand. But it appeared very soon that it was impossible to manage by hand the "combinatorial explosion" and it was obviously necessary to implement this product in a software tool. Under the supervision of Maurice Nivat, Alaiwan realized such a tool, together with some algorithms to analyze the structure of transition systems [1]. It was the preliminary version of the tool christened MEC by Maurice Nivat (it is a shortening of "Meccano", the french name of the game also known as "Erector set").

Fig. 1. A synchronized product of transition systems.

Several versions of the tool have been developed, until the last one, which is described in [4] and which differs from Alaiwan's one by a much larger set of properties which can be checked on a transition system, so that MEC can be considered as a model-checker for a logic described by Dicky [9].

This tool was used to design and check the embedded software of a household electricity meter, as reported in [5] (Fig. 1 is a picture of this meter). Did Maurice

Nivat imagine, when he published his paper [14], that, some years later, synchronized products of transition systems would be installed in thousands and thousands of homes?

More recently, the synchronous product has been introduced as one of the conceptual bases of the AltaRica formalism for describing complex industrial systems [6]. This formalism is intended to serve as an input language for a software workbench devoted to risk analysis. In order to comply with industrial needs, the synchronization product has been extended in several ways. Some of these extensions are just notational short cuts, others consist in minor modifications of the basic notion of synchronization product, such as the introduction of priorities on the synchronization vectors. Two of them offer a special interest in the perspective of this review.

First, the complex systems are usually designed in a hierarchical way, and direct interactions between components at a different level of the hierarchy are forbidden for modularity reasons (replacement of a subsystem by another one functionally equivalent). Only indirect interactions are allowed. To implement this constraint, we have reintroduced in the synchronization mechanism, the controller that was so early and unwisely given up, but with an additional role: to provide an interface between the lower and upper levels of the hierarchy.

Second, we had to renounce the dogma that states play no role in the synchronization mechanism. In fact, in the systems we are interested in, in contrast with purely software systems, some components can communicate and exchange information on their respective states even in the absence of events or actions (just think of a turbine and a steam generator connected by a pipe). The solution proposed in AltaRica is to split the state of a transition system into two parts, a control part, which remains hidden, and a visible part whose value depends on the value of the control part. For instance, let us imagine that there is a valve between the generator and the turbine. It can be in one of the two control states `open` or `closed`. The visible part of a state is the pair of steam pressures P_g and P_t at the generator and the turbine side of the valve, which can be `high` or `low`. If the valve is `open` then $P_g = P_t$. If it is `closed` then $P_t = \texttt{low}$. (This example explains why the visible parts of a state are named "flows".) In the generator–valve–turbine system, the information exchanged between the components are expressed by stating that P_g is equal to the pressure delivered by the generator, and that the pressure on the turbine is P_t. An action can modify only the control part of the state, while the visible part is modified either as a consequence of the modification of the control part of the state, or because of interactions with other components (for instance P_t can go from `high` to `low` if the valve is closed or if the generator fails).

Although this synchronization mechanism can be expressed by synchronization vectors (but at the cost of dramatically increasing the alphabet of actions of each component), it is more natural and more usable to describe complex computerized systems whose components interact in two very different ways, by physical mechanisms (pressure, voltage, etc.) as well as by electronic control devices (control program of a valve).

The base of any concurrency theory is the definition of concurrent events. The synchronization product of Maurice Nivat is the way he formalized his view of this central question. Like many other notions introduced by him in computer science, it is both simple and powerful, because it directly addresses the core of the problem.

Since this notion is not "marvellously intricate" it could not be the starting point of a large set of theorems, although it suggested or renewed many problems in language and automata theory. However, along the time, it revealed itself as a very fundamental notion on which many works about concurrent systems and their verification are based.

References

[1] H. Alaiwan, Algorithmes d'analyse des automates finis et applications aux problèmes de synchronisation, Ph.D. Thesis, Université Paris VII, 1983.
[2] A. Arnold, Transition systems and concurrent processes, in: G. Mirkowska, H. Rasiowa (Eds.), Mathematical Problems in Computation Theory, Banach Center Publications, Vol. 21, Polish Scientific Publishers, 1988, pp. 9–20.
[3] A. Arnold, Finite Transition Systems. Semantics of Communicating Sytems, Prentice-Hall, Englewood Cliffs, NJ, 1994.
[4] A. Arnold, D. Bégay, P. Crubillé, Construction and Analysis of Transition Systems with MEC, AMAST Ser. Comput., Vol. 3, World Scientific, Singapore, 1994.
[5] A. Arnold, D. Bégay, J.-P. Radoux, The embedded software of an electricity meter: an experience in using formal methods in an industrial project. Sci. Comput. Programming 28 (1997) 93–110.
[6] A. Arnold, G. Point, A. Griffault, A. Rauzy, The AltaRica formalism for describing concurrent systems, Fund. Inform. 40 (2000) 109–124.
[7] A. Bergeron, A unified approach to control problems in discrete event processes, RAIRO-Inform. Théor. 27 (1993) 555–573.
[8] L. Boasson, M. Nivat, Centers of languages, in: Proceedings of the Fifth GI Conference in Computer Science, Lecture Notes on Computer Science, Vol. 104, 1981, pp. 245–251.
[9] A. Dicky, An algebraic and algorithmic method of analyzing transition systems, Theoret. Comput. Sci. 46 (1986) 285–303.
[10] C.A.R. Hoare, Communicating sequential processes, Commun. ACM 21 (1978) 666–677.
[11] R.P. Kurshan, Modeling concurrent processes, in: Proceedings of Symposia on Applied Mathematics, vol. 31, American Mathematical Society, Providence, RI, 1985, pp. 45–57.
[12] P.E. Lauer, P.R. Torrigiani, M.W. Shields, COSY—a system specification language based on paths and processes, Acta Inform. 12 (1979) 109–158.
[13] R. Milner, Processes: a mathematical model of computing agents, in: H.E. Rose, J.C. Shepherdson (Eds.), Proceedings of Logic Collo., 1973, pp. 157–173.
[14] M. Nivat, Sur la synchronisation des processus, Rev. Techn. Thomson-CSF 11 (1979) 899–919.
[15] M. Nivat, A. Arnold, Comportements de processus, in: Collo. AFCET Les Mathématiques de l'informatique, 1982, pp. 35–68.
[16] P.J.G. Ramadge, W.M. Wonham, Supervisory control of a class of discrete events processes, SIAM J. Control Optim. 25 (1987) 206–230.
[17] P.J.G. Ramadge, W.M. Wonham, The control of discrete event systems, in: Proceedings of the IEEE, Vol. 77, 1989, pp. 81–98.