Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 209 (2008) 149–164
www.elsevier.com/locate/entcs

Reflections on the Future of Concurrency Theory in General and Process Calculi in Particular

Hubert Garavel[1]

INRIA
Centre de recherche Rhne-Alpes – VASY team
655, avenue de l'Europe
Montbonnot
38334 St Ismier cedex
France

Abstract

In this article we review the current state of concurrency theory with respect to its industrial impact. This review is both retrospective and prospective, and naturally encompasses process calculi, which are a major vector for spreading concurrency theory concepts. Considering the achievements, but also the failures, we try to identify the causes that, so far, prevented a larger dissemination of process calculi. This suggests a new generation of formal specification languages that would combine the concurrent features of process calculi with the standard concepts present in algorithmic languages. Finally, we underline two major evolutions in the software and hardware industries that open new application domains for the concurrency theory community.

Keywords: Concurrency theory, concurrent system, critical system, formal method, formal specification, model-driven architecture, model-driven engineering, modeling, modelling, process algebra, process calculus, specification, validation, verification.

1 Introduction

This article was inspired by two challenging questions raised during two scientific meetings held in 2006:

- The first question was asked in Grenoble during an evaluation meeting of the author's research team. In essence, the question was the following: "*Why are you trying to compile process calculi? Process calculi are formalisms to study theoretical aspects of concurrency, not programming languages to describe real-life systems*".

[1] Email: hubert.garavel@inria.fr

doi:10.1016/j.entcs.2008.04.009

- The second question was asked in Palaiseau during the *LIX Colloquium on Emerging Trends in Concurrency Theory*. The question was very direct, if not provocative given the number of prestigious attendees in the audience: "*Other fields of computer science have found useful applications in industry. Does concurrency theory have similar achievements? What are your success stories? Where are your victories?*".

Apparently antagonistic, both questions address in fact the very nature of concurrency theory, its applicability, as well as the status and impact of process calculi, which are a major — if not the prime — vector for the dissemination of concurrency theory results. All these points deserve a scientific discussion, and this is what the present article is about.

It is organized as follows. Section 2 recalls some important, yet partial, achievements of concurrency theory in general, and process calculi in particular. Section 3 discusses three main issues that often prevent process calculi from being widely used in industry. Section 4 mentions two important evolutions affecting the software and hardware industries, and discusses how these evolutions provide new opportunities to concurrency theory. Finally, Section 5 gives a few concluding remarks.

2 Impact of Concurrency Theory: Achievements

Concurrency theory is a field of computer science that has been producing many deep, fundamental results. As a tentative classification, we can mention:

- *Models* to represent the behaviour of concurrent systems: Petri nets, Labelled Transition Systems, Kripke structures, event structures, bigraphs, etc.,
- *Formalisms* to specify concurrent systems at a higher abstraction level: process calculi, computer languages derived from process calculi, μ-calculus, etc.,
- *Semantics* such as Structured Operational Semantics (SOS), algebraic laws, bisimulation relations, congruence properties, etc.,
- *Algorithms* to compile, execute, and verify concurrent systems expressed using models and/or formalisms,
- *Tools* that implement (a significant subset of) the above results.

In this respect, the two questions quoted in the introduction sound like a reminiscence of the recurrent debate *pure mathematics* vs *applied mathematics*. Same as for pure mathematics (illustrated by the Bourbaki group), it exists a *pure concurrency theory* school, sometimes driven more by the abstract beauty of theoretical results than by their practical usefulness. But, as for applied mathematics (illustrated by J.J. Lions and colleagues), it also exists an *applied concurrency theory* community, that takes its inspiration from real-world problems.

From the beginning, concurrency theory has been rooted in concrete examples. It appeared at the end of the 60's as an attempt to understand problems in multiprocessor and time-sharing systems. In the 80's, these motivations were still there; for instance, Milner's book on CCS [47] was illustrated by a process scheduler that

served as a running example; also, the ISO international standard LOTOS [34] was designed to specify communication protocols and services formally.

A good argument supporting the "applicability" of concurrency theory is the impressive number of software tools developed during the past decades, such as CADP [22], the Concurrency Workbench [8,57] and its American descendants [9,10], FDR2 [38], FSP [39], LOEWE [37], the LOTOSPHERE toolset [5], the μCRL toolset [11,59], to mention only a few.

These tools have been used to model and analyze numerous real-life systems. In many cases, they allowed to discover interesting — sometimes, unexpected — properties, thus leading to better understood, more reliable systems. Taking the example of the CADP toolbox, which we will use as a guiding thread throughout this article, there are nearly one hundred case-studies performed using CADP and companion tools [2].

When dealing with real-life systems, concurrency theory has clearly won some battles. The progresses might have been slow, but they have been real. This can be seen, for instance, by considering the evolving complexity of the problems tackled using CADP; over the years, the enhancements brought to this toolbox allowed to push away the state explosion limitations, thus allowing to study more complex systems, as well as systems modelled in finer detail:

- In the 80's, one could analyze systems with less than 100 lines of LOTOS.
- In the 90's, one could analyze systems with less than 1,000 lines of LOTOS.
- In the 2000's, one can analyze systems with less than 10,000 lines of LOTOS.

As part of the most recent case-studies — and because the question of "success stories" for concurrency theory was asked explicitly during the *LIX Colloquium on Emerging Trends in Concurrency Theory* — we should mention the FORMALFAME project [3] between BULL and INRIA, in which crucial parts of the FAME multiprocessor architecture used in BULL's NOVASCALE high-end servers and TERA10 machine (one of Europe's most powerful supercomputers) have been modelled using LOTOS and verified using CADP.

More generally, there are many research teams — especially in Europe — that are also applying theoretical concurrency to industrial problems. Based on the experience acquired in many case-studies, the typical methodology can be described by the three following steps:

(i) *Find a company that designs concurrent systems with a strong need for reliability.* This step is rather easy, as there are more such companies than research teams in concurrency theory. Typical application areas are communication protocols, distributed systems, hardware architectures, embedded systems, security protocols, etc.

(ii) *Select a concrete system under design by this company, model the salient parts of it formally using your favourite process calculus, and analyze this system*

[2] This list is available from `http://www.inrialpes.fr/vasy/cadp/case-studies.html`.

[3] See `http://www.inrialpes.fr/vasy/dyade/formalfame.html` for details.

using software tools for rapid prototyping, testing, verification, performance evaluation, etc.

Such a collaboration with industry can be done in two ways. The modelling task can be done by computer scientists within academia (we call this the *Dutch style*, as this seems to be the usual approach for our colleagues working in Amsterdam, Eindhoven, Twente, etc.), or it can be done within the company itself (we call this the *French style*, as the author's research team prefers this type of collaboration).

Both styles have their respective merits. The Dutch style ("*We model it for you*") is easier when starting collaborations; it relieves the company from learning a process calculus and how to use the associated tools; it allows a fast and proper modelling, as computer scientists usually avoid the mistakes and delays typically observed with novice specifiers lacking experience in formal methods. On the contrary, the French style ("*Learn and model it yourself*") is more demanding from the industrial partner, as it requires industry engineers to learn about process calculi. However, it has three advantages: the company can keep its confidential designs internally; computer scientists can focus on long-term research issues rather than spending their time investigating one particular industrial system; the collaboration, if successful, is likely to be pursued once the company has acquired a sufficient insight in formal methods.

(iii) *Reuse the feedback obtained from studying this real-life system to improve languages, algorithms, and tools, or even to start a new line of research.* Such a virtuous confrontation to real-world problems characterizes the applied concurrency approach. For instance, this drove the CADP developers into inventing new concepts of practical interest, such as efficient compiling algorithms for process calculi [16,24], μ-calculus formulas extended with data [42,41], parameterized boolean equation systems [42,41], generic programming interfaces for on-the-fly verification [18], scripting languages for compositional verification [20], real-time monitoring for distributed state-space generation [23], etc.

3 Impact of Concurrency Theory: Some Failures

So far, in spite of the achievements evoked in Section 2, the impact of concurrency theory has not been as strong as other fields of computer science. Concretely, most software engineers are not aware of the results brought by concurrency theory, and no mainstream language for specification or programming is based on the ideas behind process calculi.

For instance, the community working on object-oriented languages was more successful in spreading its ideas to a larger audience. Today, several mainstream programming languages (such as JAVA, C++, and C#) are based on the concepts developed by this community. Similarly, the community working on so-called "semi-formal" (actually, informal) methods was more successful in promoting formalisms such as UML, AADL, SYSML, etc.

There are at least two factors that prevent a wide dissemination of concurrency

theory:

- A first factor is certainly the mathematical style found in most concurrency theory articles and textbooks. Mathematical notations are certainly needed to deal with involved semantic points, but their usage beyond necessity should be avoided, not to create an entry barrier for newcomers.
- A second factor is the central role played by process calculi. They are required to model systems formally, and to apply many results of concurrency theory (such as congruence results that allow a complex system to be verified compositionally using a "divide and conquer" approach). However, process calculi are not easily accepted by industry, for at least three main reasons, which we discuss hereafter.

3.1 Issue 1: Fragmentation

A first obstacle to the dissemination of process calculi is the existence of several calculi, similar in their principles but incompatible in their details. This creates confusion for industrial users, who are unsure about which language to adopt.

This fragmentation issue was recognized as soon as the mid 80's when a ISO standardization committee led by Ed Brinksma undertook the design of LOTOS by merging the best features of CCS [47,48], CSP [31,6], and CIRCAL [44] into a unique language. Although the standardization of LOTOS in 1989 provided a foundation for ambitious research projects (such as the European projects SEDOS, LOTOSPHERE, SPECS, EUCALYPTUS, etc.), LOTOS failed to supersede or otherwise replace preexisting process calculi.

In the United Kingdom, for instance, the CCS and CSP communities continued working with their favourite calculi. It is worth noticing, however, that (according to the publications) the definition of CSP gradually evolved to become quite similar to LOTOS, at least for the untimed aspects (see e.g. [12]), still with slightly different notations. Additionally, the principles of CSP gave birth to new languages such as FDR2 and OCCAM [33] supported by compilers and verification tools.

Also, LOTOS was never recognized as a standard in the Netherlands, except in Twente. Alternative languages — similar to LOTOS but incompatible with it — were launched several years after the LOTOS standard was issued, e.g. PSF [43] or μCRL [26].

This phenomenon is by no means specific to the concurrency theory community: a similar fragmentation occurred for almost every computer language. Another reason behind fragmentation is the fact that the usage of formal methods in industry is essentially a service activity; as such, it obeys a proximity criterion, meaning that an industrial company using formal methods will naturally co-operate with academic experts located in its neighbourhood. For this reason, English companies tend to use CSP, Dutch companies tend to use μCRL, French companies working with INRIA tend to use LOTOS, etc.

From an economical point of view, such a market fragmentation into "national" language communities is probably not optimal given the high costs of training and tool development. On the long term, one unique language (or a few languages only)

will probably emerge. On the short term, remedies against fragmentation can be taken, namely:

- *Source-to-source translators between different process calculi*: for instance, the VASY team of INRIA is studying translators from CHP to LOTOS [52], from FSP [51] to LOTOS, and from FDR2 to LOTOS; these translators aim at reusing for CHP, FSP, and FDR2 the verification functionalities of the CADP toolbox [22] that was initially developed for LOTOS.
- Software gateways between different tool environments: for instance, the SEN2 team of CWI and the VASY team of INRIA have developed connections between CADP and the μCRL toolset, thus allowing to use the CADP model checkers on μCRL specifications, as well as applying the μCRL minimizers to labelled transition systems generated from LOTOS specifications.

3.2 Issue 2: Lack of Expressiveness

The "classical" process calculi (namely LOTOS, CSP/FDR2, and μCRL) allow to describe concurrent processes with complex data structures. Experience gained from numerous case-studies indicates that these calculi are sufficiently expressive for many industrial problems. However, there are certain classes of problems for which a greater expressiveness is needed:

(i) *Real-time aspects:* The behaviour of certain systems depends not only on their interactions with their environment, but also on the quantitative amount of elapsed time. To model such systems adequately, several "timed" process calculi have been proposed, in addition to timed automata, timed Petri nets, etc.

(ii) *Performance aspects:* For certain systems, one needs to model not only the functional behaviour, but also quantitative aspects, such as probabilities, rates, or throughputs of certain events. For this purpose, "probabilistic" and "stochastic" process calculi have been proposed, e.g. [2,3,29,30], as well as means to reuse and extend "classical" process calculi for performance evaluation, e.g. [28,19].

(iii) *Mobility aspects:* There are systems in which concurrent processes and communication links between processes cannot be determined statically, but evolve dynamically. All the aforementioned "classical" process calculi offer some form of dynamicity, since they allow concurrent processes to be started and terminated dynamically (this is usually achieved by using recursion through parallel composition operators). As regards communication links, these calculi support value-passing communications as well as n-ary synchronization (broadcast) between processes. However, communication links between processes remain fixed statically.

A simple extension was proposed in [25], which suggests the introduction of an "n among m" parallel composition operator. This operator allows the dynamic creation or deletion of communication links between processes, simply by exchanging data values (e.g., integers) within a pool of concurrent processes.

For instance, [25] shows how a "server" process and a "client" process can discover each other's existence dynamically and establish communication with the initial help of a "trader" process. A key point of this approach is that it remains compatible with model-checking verification tools based on finite state space enumeration.

A fundamental step forward was made with the introduction of "mobile" calculi (such as the π-calculus [45,46,54], the join calculus [14,15], etc.), which allow processes and channel names to move from one location to another. This further evolved with the even more general model of bigraphs [49]. These approaches bring full mobility, at the expense of introducing higher-order features that make state spaces potentially infinite. For this reason, verification of mobile calculi specifications often relies on theorem proving, as finite state model checking is not an option.

Considering the variety of case-studies tackled using different formalisms, it is clear that timed extensions have useful results for scheduling, that stochastic extensions are useful for performance and dependability studies, and that mobility extensions are useful to model evolving systems such as business processes. At the same time, given the number of industrial problems that do not require such extensions, it is clear that research on enhancements to "classical" process calculi remains an important research topic that should not be neglected.

3.3 Issue 3: Lack of User-Friendliness

If expressiveness is not so much an issue in many cases, user-friendliness is a major concern. It is generally admitted that the dissemination of process calculi is limited by their steep learning curve, which is mostly due to the fact that process calculi are significantly different from mainstream programming languages. This is a key problem in the aforementioned "*Learn and model it yourself*" approach. Formal specification is a time-consuming activity and industry lacks experts trained to formal methods.

The lack of user-friendliness, which greatly hampers the dissemination of process calculi, is the result of unappropriate technical decisions. Although process calculi were a scientific breakthrough, their design was not free from mistakes. We briefly evoke two main mistakes, which are directly related to the two questions raised in the introduction of the present article.

(i) There is a frequent confusion between a *calculus* and a *language*. A calculus is a minimal language intended to the theoretical study of concurrency; it should have a minimal number of primitive constructs (following Occam's razor principle) to reduce the length of induction proofs.

To the contrary, a computer language is intended to specify real, complex systems. Such a language does not need to be minimal as a primary goal; instead, it should satisfy good properties, such as *readability*, which may require to add extra keywords in the syntax, and *conciseness*, which may require to enrich the language with non-primitive constructs for handling common situa-

tions.

There have been many attempts to use a calculus in place of a language [4]. This is clearly a mistake, since a programming language must be accepted by software and system programmers as a daily working tool.

(ii) There is another design mistake, which we could call the *total algebra ideology*. In this school of thinking, every language feature must be algebraical: a concurrent language is an algebra; a concurrent program is an algebraic term; the semantics of process operators is defined by algebraic laws; data values handled by concurrent programs are algebraic terms; functions to handle data values are defined by user-specified algebraic equations; etc.

Such a language might be appealing to some mathematicians but, considering the history of computing, it seems that the mass of software programmers tend to reject algebraical languages. In that respect, all attempts to reduce computer languages to algebra have failed with a remarkable consistency.

For instance, the APL language [36], despite an initial sucess, steadily declined since 1980, surviving only in a tiny community. Interestingly, APL concepts had little impact, contrary to its competitors FORTRAN and ALGOL, most features of which have been retained in modern programming languages.

The same could be said of algebraic data types and equational specifications, the popularity of which has gradually decreased after an early period of high scientific interest. Algebraic data types may still exist in some specification languages (process calculi such as LOTOS and μCRL, of course, but also languages used in theorem provers) but they tend to be gradually supplanted by functional languages (as in FDR2, for instance).

A simple example illustrates the practical consequences of these two mistakes. It is interesting to observe how process calculi answer the need for conditionals. Most process calculi do not provide the classical "**if-then-else**" construct; instead, they use guard operators, such as, in CCS and LOTOS, "$[V] \rightarrow B$", which expresses that behaviour B can only be executed if condition V is true, or, in ACP and μCRL, "$B_1 \lhd V \rhd B_2$", which expresses that, if condition V is true then behaviour B_1 is executed or else behaviour B_2 is executed. These various guard operators have an algebraic syntax — they can be seen as unary or binary operators over processes — and a minimal semantics — only one or two SOS rules. However, they are inadequate in practice, since the CCS/LOTOS solution requires two occurrences of V to express a trivial "**if** V **then** B_1 **else** B_2" conditional, while the ACP/μCRL solution forces a counter-intuitive ordering of V's and B's when expressing nested conditionals such as "$(B_1 \lhd V_2 \rhd B_2) \lhd V_1 \rhd B_3$". We believe that the correct solution would be to have standard, properly bracketed conditionals of the form:

if...**then**...[**elsif**...**then**...]*...**else**...**endif**

even at the expense of having a non-algebraic syntax and more SOS rules (see [17,21] for a discussion).

[4] On the opposite, LOTOS was probably the first example of a language specifically designed not to be a calculus

The conclusion is clear: standard programming constructs ("**if-then-else**" and "**case**" conditionals, "**for**" and "**while**" loops, etc.) exist by themselves and should not be reduced to algebra. In essence, seeking for a minimal semantics and/or an algebraic flavour does not lead to optimal design choices. The shape of a concurrent language (i.e., the language constructs presented to end-users) should be based first on ergonomy considerations, following the lessons learnt from the history of computer languages.

4 Two Opportunities for Concurrency Theory

Away from the past, the context of computer science is changing rapidly, and its recent evolutions offer two new opportunities to revive the interest in concurrency theory.

4.1 Opportunity 1: Models Everywhere

From the beginning, concurrency theory has been based on *models*, i.e., descriptions of real-world systems, which are simplified [5] to retain only those aspects pertinent to the study of concurrency. Quite early, these models have been made *formal*, especially with the advent of process calculi.

Although the concept of model is widely used in many branches of science and engineering (e.g., physics, biology, civil engineering, etc.), formal models of concurrency have been so far confined to specific topics (formal methods, model-checking) with little impact on computer science in general.

This situation is about to change, as other computer science communities (software engineering, distributed systems, object-oriented languages, etc.) also recognized the benefits of modelling for software and systems development, and started to promote it actively. This led to the design of UML (*Unified Modelling Language*), followed by the MDA (*Model Driven Architecture*) and MDE (*Model Driven Engineering*) methodologies for software development. These methodologies lay the emphasis on *models* (real-world system abstractions seen as first-class entities) and *transformations* between models (which unifies several pre-existing concepts, such as translation and refinement). Compared to the approaches developed by the concurrency theory community, these methodologies exhibit three main differences:

(i) The MDA/MDE specification languages are mostly *graphical*, based on the underlying assumption that graphical languages are more easily accepted by industry. On the contrary, process calculi (but a few exceptions) have a textual syntax, as this was deemed to be sufficient for the study of concurrency.

(ii) The MDA/MDE specification languages can be *domain specific* in the sense that each application domain can define (or customize) its own language to fit its own needs. To the contrary, concurrency theory seeks for generality and is independent from any particular application area, in the same way as

[5] nowadays, one would say "abstracted"

linear algebra is the same for, e.g., both automotive and aerospace industries. Similarly, behind each process calculus (except a few calculi, such CIRCAL and CHP [40,50], which explicitly target at hardware circuits), there was an implicit motivation that this calculus should be general-purpose and expressive enough to model concurrent systems of various domains.

(iii) The MDA/MDE specification languages are *informal* (or *semi-formal*, which is not very different). For these languages, only the syntax is defined formally (using the notions of "meta-metamodel", "metamodel" and "model" which reformulate, in a graphical setting, the well-known concepts of BNF grammar, language, and program, respectively). There is little semantics in the MDA/MDE methodologies. Static semantics constraints (such as identifier binding and type checking) are expressed using semi-formal languages such as OCL (*Object Constraint Language*), which are not intended to allow computer-aided analysis. Dynamic semantics is totally ignored, in a sharp contrast with process calculi, for which dynamic semantics is the central topic, while syntax and static semantics aspects are kept to the minimum.

This suggests that the MDA/MDE methodologies could be enhanced by results from the concurrency theory, e.g., operational semantics, axiomatic semantics, behavioural typing, etc.

4.2 Opportunity 2: Asynchrony Everywhere

Because concurrency is a difficult matter, its usage has been often limited to a few specific areas of computing, such as multi-user operating systems, databases, scientific computing, networking, and multiprocessor architectures. Today, the list of those areas is in expansion because of the global connectivity brought by the Internet (most machines and embedded devices are now interconnected), which opens new usages, such as Web services, mobility applications, etc.

At the same time, another major revolution is taking place on the front of multiprocessors. For long, hardware design has been mostly "synchronous", meaning that all parts of a circuit would be synchronized by one single, central clock. In concurrency theory, this concept gave birth to synchronous process calculi, such as SCCS and ESTEREL.

There are good reasons for synchronous designs: synchrony is easier to master than asynchrony (thus making it easier to design correct circuits); it enforces determinism (thus enabling nonregression testing); it is well-supported by industrial computer-aided design tools. The synchronous paradigm has been very successful: it allowed to improve the performance of microprocessors by increasing regularly clock frequencies, thus enforcing Moore's law.

At the same time, there have been some attempts to build "asynchronous" circuits using a different paradigm (absence of central clock). Even when successful, these attempts remained marginal compared to the wide success of the synchronous paradigm. The situation is changing, due to several factors:

- The reduction of energy consumption is a key issue for the autonomy of embed-

ded systems. As the silicon surface of circuits increases, the electric consumption increases too, and it is common that 50% or more of the energy is used only to propagate the global clock in every part of the circuit. Also, the regularity of synchronous designs makes them vulnerable to spying (e.g., by observation of their electromagnetic emissions), which is a drawback for secure applications such as cryptography. For these reasons, the synchronous design paradigm is being questioned, and there is a growing interest in asynchronous logic for circuit design [27,13,56]. However, asynchronous logic is more error-prone than synchronous logic (with a higher risk of deadlocks for instance), thus giving a crucial role to functional verification. In this respect, process calculi and model checking tools can greatly contribute to the design of correct asynchronous circuits, e.g. [60,58,53].

- At a higher level, computer architectures are evolving to allow reductions in silicon surface and cost. Many circuits (such as arithmetic co-processors, digital signal processors, etc.) that used to be external to the microprocessor (i.e., on the motherboard) are now internal. The hardware buses themselves are moving from the outside to the inside of the microprocessor. Therefore, it is no longer efficient, nor even feasible, to have a unique clock to synchronize all these various hardware subsystems produced separately by different companies. Given that these subsystems are still designed under the sychronous paradigm, the least that must be done is to allow them to work asynchronously according to the GALS (*Globally Asynchronous, Locally Synchronous*) paradigm [7]. Again, languages and tools derived from concurrency theory can help to study such architectures, e.g. [1].
- As regards microprocessors, the limits of Moore's law are about to be reached. Due to physical/electrical barriers, it will be no longer possible to increase the frequencies of processor clocks as before. The only solution found by hardware makers to continue delivering better performance is provide several processing units (called "*cores*") inside the same microprocessor. Thus, the competition between microprocessors will no longer be expressed in terms of clock frequencies, but in the number of cores (currently, 2, 4, or 8, and this number is expected to grow rapidly).

  This should be a turning point in software development too. Because of the upper limit on clock frequencies, sequential programs will not run faster on multi-core processors than they run today on single-core processors (they might even run a bit slower). Only parallel programs will take advantage of multi-core processors, a situation that Tony Hoare, in his lecture given at the *LIX Colloquium*, characterized as follows: "*From now on, software programmers will be responsible for maintaining Moore's law*". This will require major adaptations of existing applications, from a software industry that, so far, has been relying on hardware designers to get more performance and will be soon confronted to the difficulties of wide-scale concurrency.

5 Conclusion

Research in concurrency theory has produced a vast corpus of deep, valuable results. However, these results are only known, understood, and applied by a small fraction of computer scientists. As a consequence, their practical impact is not as strong as it could be, in spite of successful attempts at using process calculi to model and verify industrially critical systems. We believe, however, that concurrency theory still has an important role to play, and that the present context (see Section 4) is ideal for this:

- Software systems modelling is becoming a standard industrial practice. There is an opportunity for the concurrency theory community to incorporate its results into the model-driven approaches, bringing semantics foundations and trying to replace informal models with formal ones.
- Parallel computing facilities will be present everywhere, ranging from embedded devices and personal computers with multi-core processors to clusters and grids. Many software applications will need to be rewritten to fully benefit from these new architectures.

Combined to increasing demands for hardware and software reliability, these evolutions promise a bright future for the theoretical concurrency community. However, this will only happen if this community improves the dissemination of its theoretical results and devotes enough attention to applications. Otherwise, it is to be feared that results not transferred in proper time will be forgotten and reinvented somewhere else. To make such an "applied concurrency theory" agenda possible, three lines of actions should be considered:

(i) The results accumulated by concurrency theory should be revisited from a Darwinian perspective: after the *mutation* phase, which gave birth to thousands of results, there should come a *selection* phase, in which the most useful results will be selected and presented to a larger audience. For instance, do we need so many process calculi? And do we really need to define fifty or more behavioral equivalences while we know that only three or four of them are sufficient in practice?

(ii) The results of concurrency theory are not easily applicable to languages based on threads, shared variables, locks, and semaphores, such as Java. It is therefore essential to maintain and increase the dissemination of process calculi. For doing so, one needs better languages than today. As discussed in Section 3, one should avoid unecessary fragmentation between multiple languages, and address the lack of user-friendliness (not simply expressiveness) — a major criterion for industrial acceptance.

In many formal specifications, it appears that only 20% of the code is devoted to concurrent aspects, while 80% of the code deals with standard data type definitions and sequential data manipulation. Therefore, process calculi should move away from a purely algebraical framework and get closer to mainstream imperative programming languages. Ideally, one should combine the classical

concepts of structured programming (for data types and sequential computations) with the key features of process calculi (parallel composition, action hiding, formal semantics, compositionality, and congruence results). This approach would certainly reduce learning time for novice specifiers, since 80% of formal specifications would be similar to sequential programs, while only 20% would require specific training in concurrency theory. These ideas were pushed forward during the design of E-LOTOS [35], and partly integrated into this international standard. Also, recent languages such as LOTOS NT [55], CHI [32], and MODEST [4] seem to follow the same principles.

(iii) The times have gone, where formal methods were primarily a pen-and-pencil activity for mathematicians. Today, only languages properly equipped with software tools will have a chance to be adopted by industry. It is therefore essential for the next generation of languages based on process calculi to be supported by compilers, simulators, verification tools, etc. This also applies to new models for concurrency, such as mobile calculi and bigraphs. The research agenda for theoretical concurrency should therefore address the design of efficient algorithms for translating and verifying formal specifications of concurrent systems.

References

[1] Garth Baulch, David Hemmendinger, and Cherrice Traver. Analyzing and Verifying Locally Clocked Circuits with the Concurrency Workbench. In *Great Lakes Symposium on VLSI (GLSVLSI'95)*, pages 144–147. IEEE Computer Society, 1995.

[2] Marco Bernardo and Roberto Gorrieri. A Tutorial on EMPA: A Theory of Concurrent Processes with Nondeterminism, Priorities, Probabilities and Time. *Theoretical Computer Science*, 202(1–2):1–54, 1998.

[3] Marco Bernardo and Roberto Gorrieri. Corrigendum to "A Tutorial on EMPA: A Theory of Concurrent Processes with Nondeterminism, Priorities, Probabilities and Time". *Theoretical Computer Science*, 254(1–2):691–694, 2001.

[4] Henrik Bohnenkamp, Pedro R. d'Argenio, Holger Hermanns, and Joost-Pieter Katoen. MoDeST: A Compositional Modeling Formalism for Real-Time and Stochastic Systems. *IEEE Transactions on Software Engineering*, 32(10):812–830, 2006.

[5] T. Bolognesi, J. van de Lagemaat, and C. Vissers, editors. *LOTOSphere: Software Development with LOTOS*. Kluwer Academic Publishers, 1995.

[6] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the ACM*, 31(3):560–599, July 1984.

[7] Daniel Marcos Chapiro. *Globally-Asynchronous Locally-Synchronous Systems*. Doctoral Thesis, Stanford University, Department of Computer Science, October 2984.

[8] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench. In J. Sifakis, editor, *Proceedings of the 1st Workshop on Automatic Verification Methods for Finite State Systems (Grenoble, France)*, volume 407 of *Lecture Notes in Computer Science*, pages 24–37. Springer Verlag, June 1989.

[9] Rance Cleaveland, Philip M. Lewis, Scott A. Smolka, and Oleg Sokolsky. The Concurrency Factory: A Development Environment for Concurrent Systems. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th International Conference on Computer Aided Verification CAV'96 (New Brunswick, New Jersey, USA)*, volume 1102 of *Lecture Notes in Computer Science*, pages 398–401. Springer Verlag, August 1996.

[10] Rance Cleaveland, Tan Li, and Steve Sims. The Concurrency Workbench of the New Century (Version 1.2). User's manual, July 2000.

[11] D. Dams and J. F. Groote. Specification and Implementation of Components of a μCRL Toolbox. Technical Report Logic Group Preprint Series 152, Utrecht University, December 1995.

[12] J. W. Davies and S. A. Schneider. A Brief History of Timed CS. *Theoretical Computer Science*, 138(2):243–271, February 1995.

[13] A. Davis and S.M. Nowick. An Introduction to Asynchronous Circuit Design. UUCS 97-013, University of Utah, Department of Computer Science, September 1997.

[14] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages POPL'96 (St. Petersburg Beach, Florida, United States)*, pages 372–385, 1996.

[15] Cédric Fournet and Luc Maranget. The Join-Calculus Language – Release 1.05. Documentation and User's Manual, September 2000.

[16] Hubert Garavel. *Compilation et vérification de programmes LOTOS*. Thèse de Doctorat, Université Joseph Fourier (Grenoble), November 1989.

[17] Hubert Garavel. A Wish List for the Behaviour Part of E-LOTOS. Rapport SPECTRE 95-21, VERIMAG, Grenoble, December 1995. Input document [LG5] to the ISO/IEC JTC1/SC21/WG7 Meeting on Enhancements to LOTOS (1.21.20.2.3), Liège (Belgium), December, 18–21, 1995.

[18] Hubert Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In Bernhard Steffen, editor, *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98 (Lisbon, Portugal)*, volume 1384 of *Lecture Notes in Computer Science*, pages 68–84, Berlin, March 1998. Springer Verlag. Full version available as INRIA Research Report RR-3352.

[19] Hubert Garavel and Holger Hermanns. On Combining Functional Verification and Performance Evaluation using CADP. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, *Proceedings of the 11th International Symposium of Formal Methods Europe FME'2002 (Copenhagen, Denmark)*, volume 2391 of *Lecture Notes in Computer Science*, pages 410–429. Springer Verlag, July 2002. Full version available as INRIA Research Report 4492.

[20] Hubert Garavel and Frédéric Lang. SVL: a Scripting Language for Compositional Verification. In Myungchul Kim, Byoungmoon Chin, Sungwon Kang, and Danhyung Lee, editors, *Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems FORTE'2001 (Cheju Island, Korea)*, pages 377–392. IFIP, Kluwer Academic Publishers, August 2001. Full version available as INRIA Research Report RR-4223.

[21] Hubert Garavel and Frédéric Lang. NTIF: A General Symbolic Model for Communicating Sequential Processes with Data. In Doron Peled and Moshe Vardi, editors, *Proceedings of the 22nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems FORTE'2002 (Houston, Texas, USA)*, volume 2529 of *Lecture Notes in Computer Science*, pages 276–291. Springer Verlag, November 2002. Full version available as INRIA Research Report RR-4666.

[22] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In Werner Damm and Holger Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification CAV'2007 (Berlin, Germany)*, volume 4590 of *Lecture Notes in Computer Science*, pages 158–163. Springer Verlag, jul 2007.

[23] Hubert Garavel, Radu Mateescu, Damien Bergamini, Adrian Curic, Nicolas Descoubes, Christophe Joubert, Irina Smarandache-Sturm, and Gilles Stragier. DISTRIBUTOR and BCG_MERGE: Tools for Distributed Explicit State Space Generation. In Holger Hermanns and Jens Palberg, editors, *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2006 (Vienna, Austria)*, volume 3920 of *Lecture Notes in Computer Science*, pages 445–449. Springer Verlag, March–April 2006.

[24] Hubert Garavel and Wendelin Serwe. State Space Reduction for Process Algebra Specifications. *Theoretical Computer Science*, 351(2):131–145, February 2006.

[25] Hubert Garavel and Mihaela Sighireanu. A Graphical Parallel Composition Operator for Process Algebras. In Jianping Wu, Qiang Gao, and Samuel T. Chanson, editors, *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification FORTE/PSTV'99 (Beijing, China)*, pages 185–202. IFIP, Kluwer Academic Publishers, October 1999.

[26] J. F. Groote and A. Ponse. The Syntax and Semantics of μCRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes'94*, Workshops in Computing Series, pages 26–62. Springer Verlag, 1995.

[27] Scott Hauck. Asynchronous Design Methodologies: An Overview. *Proceedings of the IEEE*, 83(1):69–93, January 1995.

[28] Holger Hermanns. *Interactive Markov Chains and the Quest for Quantified Quality*, volume 2428 of *LNCS*. Springer Verlag, 2002.

[29] Jane Hillston. Process Algebras for Quantitative Analysis. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*, pages 239–248. IEEE Computer Society Press, June 2005.

[30] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A Tool for Automatic Verification of Probabilistic Systems. In H. Hermanns and J. Palsberg, editors, *12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, volume 3920 of *Lecture Notes in Computer Science*, pages 441–444. Springer Verlag, March 2006.

[31] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[32] A.T. Hofkamp and J.E. Roodae. Chi 1.0 Reference Manual (Revision: 1322). Technical report, Eindhoven University of Technology, Systems Engineering Group, Department of Mechanical Engineering, February 2007.

[33] INMOS. OCCAM 2.1 Reference Manual. SGS-THOMSON Microelectronics Ltd, December 1995.

[34] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1989.

[35] ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, International Organization for Standardization — Information Technology, Genève, September 2001.

[36] Kenneth Iverson. Notation as a Tool of Thought. *Communications of the ACM*, 23(8), 1980.

[37] G. Karjoth, C. Binding, and J. Gustafsson. LOEWE: A LOTOS Engineering Workbench. *Computer Networks and ISDN Systems*, 25(7):853–874, 1993.

[38] Formal Systems (Europe) Ltd. Failure-Divergence Refinement. FRD2 User Manual, June 2005.

[39] Jeff Magee and Jeff Kramer. *Concurrency: State Models and Java Programs*. Wiley, 2006 edition, April 2006.

[40] Alain J. Martin. Compiling Communicating Processes into Delay-Insensitive VLSI Circuits. *Distributed Computing*, 1(4):226–234, 1986.

[41] R. Mateescu. Local Model-Checking of an Alternation-Free Value-Based Modal Mu-Calculus. In Annalisa Bossi, Agostino Cortesi, and Francesca Levi, editors, *Proceedings of the 2nd International Workshop on Verification, Model Checking and Abstract Interpretation VMCAI'98 (Pisa, Italy)*. University Ca' Foscari of Venice, September 1998.

[42] Radu Mateescu. *Vérification des propriétés temporelles des programmes parallèles*. Thèse de Doctorat, Institut National Polytechnique de Grenoble, April 1998.

[43] S. Mauw and G. J. Veltink. A Process Specification Formalism. *Fundamenta Informaticae*, XIII:85–139, 1990. IOS Press.

[44] G. J. Milne. CIRCAL and the Representation of Communication, Concurrency, and Time. *ACM Transactions on Programming Languages and Systems*, 7(2):270–298, April 1985.

[45] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes I. *Information and Computation*, 100(1):1–40, September 1992.

[46] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes II. *Information and Computation*, 100(1):41–77, September 1992.

[47] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[48] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[49] Robin Milner. Bigraphs as a Model for Mobile Interaction. In *Proceedings of the First International Conference on Graph Transformation ICGT'02 (Barcelona, Spain)*, number 2505 in Lecture Notes in Computer Science, pages 8–13. Springer Verlag, 2002.

[50] Marc Renaudin. *TAST Compiler and TAST-CHP Language – Version 0.6*. TIMA Laboratory, CIS Group, Grenoble, 2005.

[51] Gwen Salaün, Jeff Kramer, Frédéric Lang, and Jeff Magee. Translating FSP into LOTOS and Networks of Automata. In Jim Davies, Wolfram Schulte, and Jin Song Dong, editors, *Proceedings of the 6th International Conference on Integrated Formal Methods IFM'2007 (Oxford, United Kingdom)*, Lecture Notes in Computer Science. Springer Verlag, July 2007.

[52] Gwen Salaün and Wendelin Serwe. Translating Hardware Process Algebras into Standard Process Algebras — Illustration with CHP and LOTOS. In Jaco van de Pol, Judi Romijn, and Graeme Smith, editors, *Proceedings of the 5th International Conference on Integrated Formal Methods IFM'2005 (Eindhoven, The Netherlands)*, Lecture Notes in Computer Science. Springer Verlag, November 2005. Full version available as INRIA Research Report RR-5666.

[53] Gwen Salaün, Wendelin Serwe, Yvain Thonnart, and Pascal Vivet. Formal Verification of CHP Specifications with CADP — Illustration on an Asynchronous Network-on-Chip. In *Proceedings of the 13th IEEE International Symposium on Asynchronous Circuits and Systems ASYNC 2007 (Berkeley, California, USA)*, pages 73–82. IEEE Computer Society Press, March 2007.

[54] Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[55] Mihaela Sighireanu. LOTOS NT User's Manual (Version 2.4). INRIA projet VASY. ftp://ftp.inrialpes.fr/pub/vasy/traian/manual.ps.Z, June 2004.

[56] Jens Spars and Stephen B. Furber. *Principles of Asynchronous Circuit Design: A Systems Perspective*. European Low-Power Initiative for Electronic System Design. Kluwer Academic Publishers, 2002.

[57] Perdita Stevens. The Edinburgh Concurrency Workbench (Version 7.1). User manual, 1997.

[58] X. Wang, M. Kwiatkowska, G. Theodoropoulos, and Q. Zhang. Opportunities and Challenges in Process-Algebraic Verification of Asynchronous Circuit Designs. In *Proceedings of the Second Workshop on Globally Asynchronous Locally Synchronous Design (FMGALS'05)*, volume 146 of *Electronic Notes in Theoretical Computer Science*, pages 189–206, January 2006.

[59] A. G. Wouters. Manual for the muCRL Tool Set (Version 2.8.2). Technical Report SEN R0130, CWI, Amsterdam, 2001.

[60] Tomohiro Yoneda, Atsufumi Shibayama, and Takashi Nanya. Verification of Asynchronous Logic Circuit Design using Process Algebra. *Systems and Computers in Japan*, 28(8–9):33–43, 1997.