STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIII**, Number 2, 2008

ADVANTAGES AND DISADVANTAGES OF THE METHODS OF DESCRIBING CONCURRENT SYSTEMS

ANITA VERBOVÁ AND RÓBERT HUŽVÁR

Abstract. This paper provides a review of existing paradigms for modelling concurrent processes. First we describe in short some formal methods designed for the development of the theory of concurrency.

Because no unified theory or calculus for concurrency has showed up, we concentrate on interaction categories and their features relevant for our purposes. They are able to describe some essential features of communicating processes.

Finally we confront all these methods and point out their limitations and expressive power. We highlight some open problems with regard to reasoning about concurrent systems.

1. Description of concurrent systems by process calculi

There exists many methods for the formal description of concurrent systems. The most substantial of these paradigms is the process calculus [3]. Its pioneers were Milner a Hoare with their methods CCS [7] and CSP [5] respectively. There are also another paradigms, which describe concurrent processes and some of their properties. Here belongs for instance the π-calculus [10], the structure of events [17], Petri nets [4] and SCCS [8]. SCCS (synchronous calculus of communicating systems) is a process algebra in which processes contribute their visible activity synchronously, or in other words, in unison with a global clock. The algebra also contains operators for structuring process

Received by the editors: September 14, 2008.

2000 *Mathematics Subject Classification.* 18C10.

1998 *CR Categories and Descriptors.* F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages – *Process models*; F.4.1 [**Logics and Meanings of Programs**]: Mathematical Logic – *Proof theory* .

Key words and phrases. Category theory, Concurrent systems, Process algebra, Interaction categories.

This paper has been presented at the 7th Joint Conference on Mathematics and Computer Science (7th MaCS), Cluj-Napoca, Romania, July 3-6, 2008.

definitions, renaming and inhibiting actions and permitting nondeterministic choices of behaviour [11].

Main result of this paradigm is to develop an algebraic theory of concurrency as a foundation for structural methods for describing concurrent systems.

2. Description of concurrent systems by the π-calculus

In [9] Milner describes the π-calculus as a step towards a canonical calculus for concurrent systems. It is a minimal calculus such that all programs that are computable by a concurrent system can be encoded in it. The π-calculus hopes to play a similar role for concurrent systems to that played by the λ-calculus for sequential computation.

The π-calculus is a process algebra, similar to CCS, but is designed to model systems with dynamically changing structure: that is, the links between the components of a system can vary dynamically during the evolution of the system. This property, which is called *mobility*, can at best be modelled indirectly in established process algebras.

The π-calculus allows channel names to be passed as values in communications. In fact the π-calculus combines the concepts of channel names, value and value variables into a single syntactic class: *names*. The π-calculus is not a higher order calculus: it is only accesses to agents that are being passed in communications, not the agents themselves. The passing of agents as parameters in communications is undesirable since agents would then become replicated, and the replication of agents with state is difficult. Limiting ourselves to the passing of accesses means that we can allow certain agents only limited access to other agents, and have several agents having different access abilities to some common agent.

The main features of the π-calculus are the dynamic creation of channel names and handshake communication on these names.

3. Mathematical theory of computational paradigm

The latest established of current paradigms for the semantics of computation is *denotational semantics*. In spite of its pretensions to universality, denotational semantics has a natural slant to computational paradigm: *functional computation*. By this we mean not only functional programming languages, but the whole range of computation, where the behaviour of the program is abstracted as the computation of a function. This view of programs as functions

is built into the fundamental mathematical framework, which was denotational semantics found on: a category of sets for the interpretation of types, and specific functions between these sets for the interpretation of programs.

4. Category theory for modelling concurrent systems

The development of interaction categories [15] results from the limitations of the paradigms mentioned above. These paradigms have developed independently. Their separate development is considered to be the main open problem, i.e. how can we combine the functional and concurrent process paradigms with their associated mathematical support in a single unified theory. This unification is the consequence of the following investigations:

(1) in process algebras:
   - There is no typing, hence there is a need of a good *type theory for concurrent processes*.
   - Stress is laid mainly on which are these processes, rather than on what structure they have collectively.
   - There is a real *confusion* of formalisms, combinators and equivalences.
   - Their major objection is that did not appear any *generalized theory* or calculus for concurrency.

(2) in denotational semantics:
   - Denotational semantics works well not only for the description but also for the language design and programming methods.

Unification of these two methods is necessary to obtain correct basis for languages connecting concurrent processes and communication with types on one hand, and higher order constructions with polymorphism on the other. It is also desirable for the foundations of suitable type systems for concurrency.

4.1. **Interaction categories.** In the categorical semantic approach, we define a category of processes [2], where we model types as objects, processes as morphisms, and interaction as morphism composition.

Once this structure of typed arrows closed under composition has formulated, then a great amount of further structure is determined up to isomorphism.

4.2. **Categorical structure of synchronous processes.** In [2] interaction categories are introduced by presentation of a canonical example, category of synchronous processes **SProc**. In general *objects* of interaction categories are

concurrent system specifications, their *morphisms* are synchronisation trees, *composition* is given by synchronous product and restriction and *identities* are synchronous buffers. The category **SProc** has a very rich structure.

More formally the *objects* of **SProc** are pairs $A = (\Sigma_A, S_A)$, where Σ_A is an alphabet of actions (labels) and $S_A \subseteq^{nepref} \Sigma_A^*$ is a safety specification. Hence, a safety specification is a non-empty and prefix-closed set of traces over A, which represents a linear-time safety property.

A process p of type A, written $p : A$, is a synchronization tree modulo strong bisimulation, with labels from Σ_A, such that $\texttt{traces}(p) \subseteq S_A$. Following Aczel we use a representation of synchronization trees as non-well-founded sets, in which a process p with transitions $p \xrightarrow{a} q$, $p \xrightarrow{b} r$ becomes $\{(a, q)\,(b, r)\}$.

The most convenient way of defining the morphisms of **SProc** is first to define a *-autonomous structure on objects, then say that the morphisms from A to B are processes of the internal hom type $A \multimap B$. Given objects A and B, the object $A \otimes B$ has

$$\begin{array}{rcl} \Sigma_{A\otimes B} & = & \Sigma_A \times \Sigma_B \\ S_{A\otimes B} & = & \left\{\sigma \in \Sigma^*_{A\otimes B} | \texttt{fst}^*(\sigma) \in S_A \wedge \texttt{snd}^*(\sigma) \in S_B\right\}. \end{array}$$

The duality is trivial on objects: $A^\perp = A$. This means that at the level of types, **SProc** makes no distinction between input and output. Because communication is based on synchronization, rather than on value-passing, processes do not distinguish between input and output either.

The definition of $\otimes$ makes clear how are processes in **Sproc** synchronous. An action performed by a process of type $A \otimes B$ consists of a pair of actions, one from the alphabet of A and one from that of B. Thinking of A and B as two ports of the process, synchrony means that at every time step a process must perform an action at every one of its ports.

A *-autonomous category in which $\otimes$ is self-dual, i.e. such that $(A \otimes B)^\perp \cong A^\perp \otimes B^\perp$, is a compact closed category. Hence in a compact closed category $A \parr B \cong A \otimes B$. In the special case when $A^\perp \cong A$ the linear implication, defined by $A \multimap B = A^\perp \parr B$, also corresponds to $A \otimes B$. In **SProc** $A^\perp = A$, and so $A \parr B = A \multimap B = A \otimes B$.

Not all interaction categories are compact closed, but those that are, support more process constructions than those, that are not.

A *morphism* of **SProc** $p : A \to B$ is a process p of type $A \multimap B$. Since $A \multimap B = A \otimes B$, this means for the process p that it is of type $A \otimes B$.

Given $p : A \to B$ and $q : B \to C$ then we can define their *composition* $p; q : A \to C$ in the category **SProc** as follows:

$$\frac{p \xrightarrow{(a,b)} p' \quad q \xrightarrow{(b,c)} q'}{p;q \xrightarrow{(a,c)} p';q'}$$

in which matching of actions takes place in the common type B (as in relational composition), at each time step. This ongoing communication is the *interaction* of interaction categories.

The identity morphisms are synchronous buffers: whatever is received by $\mathtt{id}_A : A \to A$ in the left copy of A is instantaneously transmitted to the right copy (and vice versa – there is no real directionality). If p is a process with sort Σ and $S \subseteq^{nepref} \Sigma^*$ then the process $p \restriction S$ is defined by:

$$\frac{p \xrightarrow{a} q \quad a \in S}{p \restriction S \xrightarrow{a} q \restriction (S/a)}$$

where $S/a \stackrel{\text{def}}{=} \{\varepsilon\} \cup \{\sigma | a\sigma \in S\}$.

The *identity morphism* $\mathtt{id}_A : A \to A$ is defined by $\mathtt{id}_A \stackrel{\text{def}}{=} \mathtt{id} \restriction S_{A \multimap A}$ where the process $\mathtt{id}$ with sort Σ_A is defined by:

$$\frac{a \in \Sigma_A}{\mathtt{id} \xrightarrow{(a,a)} \mathtt{id}}$$

Since $\oplus$ is a coproduct, its dual is a product; because all objects of **SProc** are self-dual, this means that $A \oplus B$ is itself also a product of A and B – so it is a biproduct. If $p;q : A \to B$ then their non-deterministic combinator is defined by:

$$\begin{aligned} p + q &= A \xrightarrow{\Delta_A} A \oplus A \xrightarrow{[p,q]} B \\ &= A \xrightarrow{\langle p,q \rangle} B \oplus B \xrightarrow{\nabla_B} B \end{aligned}$$

where $\Delta_A \stackrel{\text{def}}{=} \langle \mathtt{id}_A, \mathtt{id}_A \rangle$ is the *diagonal* and $\nabla_B \stackrel{\text{def}}{=} [\mathtt{id}_B, \mathtt{id}_B]$ is the *codiagonal*. To make clear the definition of $+$, consider the composition $\langle p, q \rangle ; \nabla_B$. Pairing creates a union of the behaviours of p and q, but with disjointly labelled copies of B. Composing with ∇_B removes the difference between the two copies. A choice can be made between p and q at the first step, but then the behaviour continues as behaviour of p or behaviour of q. Thus we obtain the natural representation of the *non-deterministic sum* in terms of synchronisation trees in CCS.

4.3. **Categorical structure of asynchronous processes.** Category of processes **Buf** with a similar structure as interaction categories is defined in [14]. Morphisms are given by labelled transition systems representing processes in a language like CCS. These processes are asynchronous in the sense that a sender does not wait for the message delivering as in handshake mechanism of CCS. In the category **Buf**

- Objects are sets (names of channels).
- Morphisms $A \to B$ are labelled transition systems with input actions from A and output actions from B, illustrated according to weak bisimulation.
- Composition of morphisms is interaction in the form of parallel composition and restriction.
- Identities are asynchronous buffers, *i.e.* processes, which simply forward the messages, which they deliver and they do not necessarily preserve order.

We can define products as parallel composition without interaction, and **Buf** is a traced monoidal category [6], thus it provides a feedback operation, and we are able to build cycles of processes.

The category **Buf** is obtained by restricting the sets of morphisms to those processes that are buffered. In [13], axioms are given to classify those processes that behave the same when composed with a buffer, for the case when the buffer does not preserve the order of messages (as in **Buf**), and for first-in-first-out buffers. These axioms are quite strong. They require, for example, that a process can at every state do an input transition on each input channel. For first-in-first-out buffers, they require that from each state there is at most one output transition.

5. Comparison of paradigms for the description of concurrent systems

In this section we summarize the point of view of the designed paradigms.

Still is widely appreciated that the functional computation is only one, relatively restricted part of computational universe, where distributed systems, real-time systems and reactive systems do not really fit. Success of **denotational semantics** out of the area of *functional computation* is very limited.

Partly because of the absence of a good type theory, in **process algebras** has been a considerably systematic *chaos between specifications and processes. Names* in process calculi are used as corresponding names, which distinguish

these calculi syntactically and strongly from the others. For example, process algebra tend to be more abstract and specification-oriented than Petri nets, while the latter describe concurrency at a more intricate structural level.

The **π-calculus** is not higher order, unlike the λ-calculus where λ expressions (interpreted as agents) can be passed as arguments to functions and bound to variables. In the π-calculus we cannot pass processes themselves in communications or substitute them for names. We can construct implementations of functional and higher order programming languages on the basis of passing simple data items between registers and carrying out simple operations on them, where these data items function either as pointers to the code of functions or other complex data structures, or as values, instead of passing the functions and complex data structures themselves. Perhaps the most valuable aspect of the π-calculus is that it gives us an abstract, mathematical way to model this kind of computing, and so allows us to reason about such implementations in a formal way.

It is debatable whether the π-calculus can be extended in such a way as to make representations of complex constructions easier. Summation and τ-actions produce semantic difficulties, and so it might be worth investigating some other external choice operator. Even with summation and conditional guards we could not build the infinite functions and operators. The question of how best to extend the calculus in order to make it more useful therefore remains open. Similar open problem is the extension of π-calculus to include some notions of type.

Method of **formal calculus** [1] stems from the set of combinators forming a syntax. The weakness of these methods is already in the use of this set of *combinators* rather then another.

In the category **Sproc** a synchronous product is choosen to represent the interaction of processes for the following reasons:

- Buffers are taken as the identity morphisms. This is in accordance both with *synchronous* processes, where buffers are without delay – they behave like hardware wires, also in the case where are buffered processes, in which they are insensitive to delay. Also it satisfies *asynchronous* case, where identity morphisms are also synchronous buffers.
- Milner‘s synchronous calculus SCCS is very expressive. Asynchronous calculi such as CCS and CSP can be derived from SCCS. Therefore we can take synchronous interaction as a basic notion.

Instead of considering labels to be appropriate names, a typed framework [12] is used to take a more structural view of concurrent processes. Interpretations of type constructors in **interaction categories** require set-theoretic constructions on the set of labels (sorts) associated with each type. A cartesian *product* of sorts (pairing of labels) is used to express the concurrent execution of some distributed actions. *Coproduct* is used to tag actions to allow controlled choices. Multisets of actions are used to support *replication* of processes. Product, coproduct and multisets represent in the notions of linear types [16] multiplicatives, additives and exponentials respectively. In that way we can generate such a set of categorical combinators for process algebra, which is free of labels. Therefore we should use categorical combinators for the translation of functional programs in a variable-free fashion.

Interaction categories clearly distinguish processes (computational entities) and specifications (logical entities).

Hoare in CSP considers processes with one input and one output, designed to be connected in a pipeline – this is very close to the view indicated in interaction categories. The same *divergence* problem arises in the case of interaction categories as in CSP. Two conditions are defined to avoid this situation. For the process $p; q$ must hold that p have to be *left-guarded* and q *right-guarded*. In that case p cannot perform an infinite sequence of actions in its right port without doing some actions in its left port; process q is defined symmetrically. These conditions ensure that the process $p; q$ does not diverge. Therefore if we adjust this idea to interaction categories, then we require all morphisms to be left- and right-guarded, so that all composites are non-divergent.

Here we would like to compare categories **SProc** and **Buf**. In contrast to the category **SProc**, processes $A \to B$ in the category **Buf** are oriented – channels in A are input channels, these in B are output channels.

In **SProc** the identity process is a process that continually offers to do the same action on both sides of its interface — it can be seen as a buffer that immediately sends on any message it receives. Because it is *synchronous*, the receive and the send actions happen at the same time, and so it cannot be distinguished whether a message was sent through the buffer or not. In an *asynchronous* setting a buffer will not generally work as an identity for composition.

Acknowledgement

This work was supported by VEGA Grant No.1/0175/08: Behavioral categorical models for complex program systems.

References

[1] Abramsky, S. What are the fundamental structures of concurrency? we still don't know! In *Electronic Notes in Theoretical Computer Science, 162* (2006), pp. 37–41.

[2] Abramsky, S., Gay, S., and Nagarajan, R. Interaction categories and the foundations of typed concurrent programming. In *Proceedings of the NATO Advanced Study Institute on Deductive program design* (Secaucus, NJ, USA, 1996), Springer-Verlag New York, Inc., pp. 35–113.

[3] Baeten, J. C. M. A brief history of process algebra. *Theor. Comput. Sci. 335*, 2-3 (2005), 131–146.

[4] Brauer, W., Reisig, W., and Rozenberg, G., Eds. *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, 8.-19. September 1986* (1987), vol. 255 of *Lecture Notes in Computer Science*, Springer.

[5] Hoare, C. A. R. *Communicating Sequential Processes*. Prentice Hall, 1985.

[6] Joyal, A., Street, R., and Verity, D. Traced monoidal categories. *Math. Proc. Cambridge Philos. Soc. 119*, 3 (1996), 447–468.

[7] Milner, R. *A Calculus of Communicating Systems*, vol. 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.

[8] Milner, R. *Communication and Concurrency*. 1989.

[9] Milner, R. Functions as processes. In *ICALP* (1990), pp. 167–180.

[10] Parrow, J. *An Introduction to the π-Calculus, in The Handbook of Process Algebra*. Elsevier, Amsterdam, 2001, p. 479.

[11] Ross, B. J. Mwsccs: A stochastic concurrent music language. In *In: Proc. II Brazilian Symposium on Computer Music* (1995).

[12] Schweimeier, R. A categorical framework for typing ccs-style process communication. *Electr. Notes Theor. Comput. Sci. 68*, 1 (2002).

[13] Selinger, P. First-order axioms for asynchrony. In *International Conference on Concurrency Theory* (1997), pp. 376–390.

[14] Selinger, P. Categorical structure of asynchrony. *Electr. Notes Theor. Comput. Sci. 20* (1999).

[15] Verbová, A., Hužvár, R., and Slodičák, V. On describing asynchronous processes by traced monoidal categories. In *Proceedings of CSE 2008 International Scientific Conference on Computer Science and Engineering* (2008), elfa, s.r.o. Košice, pp. 99–106.

[16] Verbová, A., Novitzká, V., and Slodičák, V. From linear sequent calculus to proof nets. In *Informatics 2007, Proceedings of the Ninth International Conference on Informatics* (2007), Slovak Society for Applied Cybernetics and Informatics Bratislava, pp. 100–107.

[17] Winskel, G., and Nielsen, M. Models for concurrency. In *Handbook of Logic in Computer Science*, S. Abramsky, D. Gabbay, and T. S. E. Maibaum, Eds. Oxford University Press, 1995.

Department of Computers and Informatics, Technical University of Košice, Slovakia

E-mail address: anita.verbova@tuke.sk

Department of Computers and Informatics, Technical University of Košice, Slovakia

E-mail address: robert.huzvar@tuke.sk