Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 162 (2006) 141–145

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A Process Algebraic View of Coordination

Nadia Busi and Gianluigi Zavattaro

Dipartimento di Scienze dell'Informazione, Università di Bologna,
Mura Anteo Zamboni 7, I-40127 Bologna, Italy.
E-mail: {busi, zavattar}@cs.unibo.it

Abstract

Coordination languages have been introduced since the early 80s as programming notations to manage the interaction among concurrent collaborating software entities. Process algebras have been successfully exploited for the formal definition of the semantics of these languages and as a framework for the comparison of different coordination models.

Keywords: Coordination language, process algebra

1 Coordination Languages: an Overview

Coordination languages are a class of programming notations which offer a solution to the problem of specifying and managing the interactions among computing agents. In fact, they generally offer language mechanisms for composing, configuring, and controlling software systems made of independent, even distributed, active components.

Gelernter and Carriero introduced a programming-specific meaning of the term *Coordination* presenting the following equation [7]:

$$\text{Programming} = \text{Computation} + \text{Coordination}$$

They formulated this equation arguing that there should be a clear separation between the specification of the components of the computation and the specification of their interactions or dependencies. On the one hand, this separation facilitates the reuse of components; on the other hand, the same patterns of interaction usually occur in many different problems – so it might be possible to reuse the coordination specification as well.

A number of interesting models have been proposed and used to design, study, and compare coordination languages. Examples include tuple spaces as in Linda [11], various forms of multiset rewriting or chemical reactions as in Gamma [2],

doi:10.1016/j.entcs.2006.08.001

models based on the raising and catching of events as in SIENA [14] or JEDI [9], and models with explicit support for coordinators as in Manifold [1].

Coordination models have been classified in two main classes [12]:

(i) *Shared dataspace*: components communicate by producing, consuming, and testing for the presence of data in a shared, common repository.

(ii) *Publish/Subscribe*: communication takes place through the raising of events performed via a *publish* operation. Events are multicast to those components which have previously registered their interest via a *subscribe* operation.

Linda [11] is the most prominent representative of the family of coordination languages based on the shared dataspace model: a sender communicates with a receiver through a shared data space (called *tuple space*), where emitted messages are collected; the receiver can read the message or even remove it from the TS; a message generated by a process has an independent existence in the tuple space until it is explicitly withdrawn by a receiver; in fact, after its insertion in the tuple space, a message becomes equally accessible to all processes, but it is bound to none.

Besides the non-blocking output operation $out(a)$ (that sends the message a to the tuple space), the blocking read operation $rd(a)$ (that succeeds only if a is in the tuple space) and the blocking input operation $in(a)$ (that removes message a from the TS), Linda offers two further conditional input and read predicates, called $inp(a)$ and $rdp(a)$ [15]. These predicates check the current status of the tuple space; if the required message a is absent, the value $false$ is returned; on the contrary, if the message is found, their behavior is the same as the in/rd operation and the value $true$ is returned.

SIENA [14] and JEDI [9] are two of the most known publish/subscribe coordination languages. Conceptually, they provide a coordination service to clients. Clients use the service to advertise the information about events that they generate and to *publish* notifications containing that information. They also use the service to *subscribe* for notifications of interest. The service then notifies clients by delivering any notification of interest.

The two models provide coordination facilities by exploiting *data* and *events*, respectively. These two abstractions can be compared with respect to the following aspects: *creation*, *life-time*, and *visibility*.

The creation is non-blocking both for events and data: an agent can raise an event in each possible context and an agent can introduce a new datum in a shared repository whatever is its actual state.

A first basic difference can be observed on the lifetime: after its raising, an event plays a role in the overall system only during the multicast protocol; on the other hand, a datum remains available in the dataspace until it is explicitly withdrawn. This property is usually referred to as *generative communication* [11]: a datum, after its production, has an independent life inside the dataspace.

Concerning the visibility, it is worth to point out at least two differences between events and data. A datum can be read from any agent, even from agents not

present in the system at the time the datum was produced; this property is usually referred to as *time-uncoupling*. On the other hand, an event can be observed only by those agents which registered their interest before the raising of the event. A second observation concerns the ability to perform a destructive consumption of information: data can be removed from the dataspace, thus disallowing other agents to read it; on the other hand, an agent cannot hide an event to the other agents in the system.

The coordination language Linda was originally conceived in the 80's to program parallel computers or local area network distributed applications. In the late 90's, with the advent of wide area network distributed applications, we have assisted to a renewed interest in such a coordination language. For example, JavaSpaces [16] and TSpaces [18] are two recent coordination middlewares for distributed Java programming proposed by Sun and IBM, respectively. These proposals incorporate the main features of both the two historical groups of coordination models. Besides the typical Linda-like coordination primitives, both JavaSpaces and TSpaces provide event registration and notification. This mechanism allows a process to register interest in the future arrivals of a particular kind of data, and then receive communication of the occurrence of these events.

2 Process Algebras for Coordination

Coordination languages are usually informally defined in reference manuals or user's documentations: see e.g. Linda [15] and JavaSpaces [16]. Process Algebras have been successfully exploited as a formal basis to provide these languages with a semantics. These formalizations provided also a framework for a comparative analysis of the coordination primitives: the main outcomes are, on the one hand, the characterization of expressiveness gaps among different interpretations/implementations of the same coordination primitive and, on the other hand, the proof of (im)possibility to reduce one coordination model into another one.

As far as Linda is concerned, the first examples of process algebraic semantics are [8] and [10]. Both these proposals define a CCS-like language whose basic atomic actions are inspired by the *in*, *rd* and *out* Linda coordination primitives. The non-blocking predicates *inp* and *rdp* have been dealt with in [3].

In [4], an interesting expressiveness gap between two semantics for the process algebra introduced in [3] has been pointed out. These two semantics follow two different intuitions expressed in the Linda reference manual [15]. The former, called *ordered*, defines the output as an operation that returns when the message has reached the shared data space; the latter, called *unordered*, returns just after sending the message to the tuple space. The process algebra under the ordered semantics is Turing powerful as it permits to program any Random Access Machine. On the contrary, the process algebra under the unordered semantics is not Turing powerful. This result is achieved by resorting to a net semantics in terms of contextual nets (P/T nets with inhibitor and read arcs), and showing that there exists a deadlock-preserving simulation of such nets by finite P/T nets, a formalism where termination

is decidable.

The analysis of the expressiveness of coordination primitives started in [4] has been extended in [5] to investigate the interplay of the event notification mechanism with the classical Linda-like coordination paradigm. In particular, we focussed on the $notify$ primitive of JavaSpaces, used by a process to register interest in the incoming arrivals of a particular kind of data, and then receive communication of the occurrence of these events. We prove the existence of a hierarchy of expressiveness among the possible combinations of coordination primitives: (i) event notification cannot be encoded with only input and output operations, but (ii) it becomes encodable if also test for absence is considered; moreover, (iii) test for absence is strictly more expressive than event notification as it cannot be encoded with only input, output and event notification.

Another interesting novelty of JavaSpaces is the notion of temporary data, that is data with an associated expiration time. This notion permits to address the problem of the accumulation of outdated and unwanted information in the shared repository. Typical garbage collection algorithms, indeed, cannot be adopted in this context because there is no notion of unaccessible data. In [6], we have investigated the impact of different mechanisms for expired data collection on the expressiveness of dataspace coordination systems with temporary data.

3 Conclusion

The novel networking technologies, such as peer-to-peer overlay networks and mobile ad hoc networks, call for the definition of new coordination languages based on new interaction metaphors. For instance, peer-to-peer networks introduced the concept of flooding, i.e. the multi-hop propagation of information among neighbours, while the native interaction mechanism in mobile ad hoc networks is wireless broadcast.

Regarding mobile networks, an interesting proposal is represented by the Lime [13] shared dataspace coordination model, improved and formalized in a process algebraic style in [17]. The main idea underlying Lime is that each agent has its own dataspace. A group of physically connected agents is called a confederation. The agents in a confederation share the same logical dataspace, which is transparently constructed by merging the local dataspaces of the agents themselves.

Lime can be considered as a starting point towards the definition of a new generation of coordination languages. We believe that the experience achieved in the formalization and comparison of the traditional coordination primitives can be exploited to drive the development of new reference models for such languages.

References

[1] F. Arbab, I. Herman, and P. Spilling. An Overview of Manifold and its Implementation. *Concurrency - Practice and Experience*, 5(1):23–70, 1993.

[2] J-P. Banatre and D. Le Metayer. Programming by Multiset Transformation. *Communications of the ACM*, 36(1):98–111, 1993.

[3] N. Busi, R. Gorrieri, and G. Zavattaro. A Process Algebraic View of Linda Coordination Primitives. *Theoretical Computer Science*, 192(2):167–199, 1998.

[4] N. Busi, R. Gorrieri, and G. Zavattaro. On the Expressiveness of Linda Coordination Primitives. *Information and Computation*, 156(1/2):90-121, 2000.

[5] N. Busi and G. Zavattaro. On the Expressiveness of Event Notification in Data-driven Coordination Languages. In Proc. of *2000 European Symposium on Programming* (ESOP'00), volume 1782 of *LNCS*, pages 41–55, 2000.

[6] N. Busi and G. Zavattaro. Expired Data Collection in Shared Dataspaces. *Theoretical Computer Science*, 298: 529-556, 2003.

[7] N. Carriero and D. Gelernter. Coordination Languages and Their Significance. *Communications of the ACM*, 35(2):97–107, 1992.

[8] P. Ciancarini, K. Jensen, and D. Yankelewich. On the Operational Semantics of a Coordination Language. Volume 924 of *LNCS*, pages 77–106, 1995.

[9] G. Cugola, E. Di Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In Proc. of *20th International Conference on Software Engineering* (ICSE'98), pages 261–271, 1998.

[10] R. De Nicola and R. Pugliese. A Process Algebra based on Linda. In Proc. of *First International Conference on Coordination Models and Languages* (COORDINATION'96), volume 1061 of *LNCS*, pages 160–178, 1996.

[11] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[12] G.A. Papadopoulos and F. Arbab. Coordination Models and Languages. *Advances in Computers*, 46:329–400, 1998.

[13] G.P. Picco, A. Murphy, and G.C. Roman. Lime: Linda Meets Mobility. In Proc. *21th IEEE Int. Conf. on Software Engineering* (ICSE'99), pages 368–377, 1999.

[14] D.S. Rosenblum and A.L. Wolf. A Design Framework for Internet-Scale Event Observation and Notification. In Proc. of *6th European Software Engineering Conference* (ESEC'97), volume 1301 of *LNCS*, pages 344-360, 1997.

[15] Scientific Computing Associates. *Linda: User's guide and reference manual*, 1995.

[16] Sun Microsystem, Inc. *JavaSpaces Specifications*, 1998.

[17] M. T. Valente, B. Carbunar and J. Vitek. Lime Revisited. Reverse Engineering an Agent Communication Model. In Proc. *5th International Conference on Mobile Agents* (MA'01), volume 2240 of *LNCS*, pages 54–69, 2001.

[18] P. Wyckoff, S.W. McLaughry, T.J. Lehman, and D.A. Ford. T Spaces. *IBM Systems Journal*, 37(3), 1998.