Science of Computer Programming 115–116 (2016) 149–176

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

On the introduction of density in tuple-space coordination languages

Jean-Marie Jacquet [a], Isabelle Linden [b], Denis Darquennes [a,*]

[a] University of Namur, Faculty of Computer Science, Rue Grandgagnage 21, Namur, Belgium
[b] University of Namur, Department of Business Administration, Rempart de la Vierge 8, Namur, Belgium

ARTICLE INFO

Article history:
Received 7 March 2014
Received in revised form 14 September 2015
Accepted 12 October 2015
Available online 10 November 2015

Keywords:
Coordination
Service-oriented computing
Tuple spaces
Density
Expressivity

ABSTRACT

Coordination languages have been proved very suitable for modeling and programming service-oriented applications. In particular, those based on tuple spaces offer an elegant way of making different components of such applications interact smoothly through the deposit and retrieval of tuples in a shared space. However, in their basic form, these languages only allow one tuple to be put at a time and, when more than one tuple matches a required one, the selection is made non-deterministically. This is obviously too weak to capture popularity or quality measures, which are nevertheless central in service-oriented applications. To that end, we propose an extension of a Linda-like language aiming at promoting the notion of density and, based on De Boer and Palamidessi's notion of modular embedding, study its expressiveness. We prove accordingly that it strictly increases the expressiveness of Linda while keeping the same implementation efficiency. We also compare it with languages based on multiset rewriting, such as Gamma, and establish that, although it is less expressive, it benefits from a much more efficient scheme. Finally we study the hierarchy of the sublanguages induced by considering subsets of tuple primitives and prove that it follows that of the Linda family of languages.

1. Introduction

Service-oriented applications have become more and more available on the Internet. The rapid evolution of their demand induces a competition between them, requiring a huge adaptive capacity. Their ability to measure their popularity and their quality of services is then crucial for their evolution, as well as for their survival on the Internet.

Besides, coordination languages have been proved very suitable for modeling and programming service-oriented applications (see e.g. [6,20,27]). Among them, those based on tuple spaces offer an elegant way of making different components of such applications interact smoothly through the deposit and retrieval of tuples in a shared space. However their basic form does only allow one tuple to be put at a time and, when more than one tuple matches a required one, the selection is made non-deterministically. This is certainly too weak to grasp notions central to Internet applications such as advices and recommendation. To that end, we introduce a notion of density in tuple space languages. The intuition is that the more a tuple is present on the tuplespace, the more likely it is of interest. Moreover, by requiring tuples with a minimum level of density, one may make sure that only those with a sufficient recognition are selected.

* Corresponding author.
E-mail addresses: jean-marie.jacquet@unamur.be (J.-M. Jacquet), isabelle.linden@unamur.be (I. Linden), denis.darquennes@unamur.be (D. Darquennes).

http://dx.doi.org/10.1016/j.scico.2015.10.011
0167-6423/

Concretely, we shall enrich the tell, ask, nask and get primitives of our Linda-based language Bach with an integer referring to how many instances of a tuple we address. Consequently, `tell(t(n))` has the effect of enriching the current tuple space with n new copies of the tuple `t`. Dually, `ask(t(n))` requires the presence of at least n instances of tuples matching the tuple `t` while `get(t(n))` additionally removes them. Finally, `nask(t(n))` succeeds when less than n instances matching tuple `t` are present on the tuple space.

Smart cities provide many interesting examples of the interest of our extension. To start with, let us imagine that we want to build an application allowing to select taxi drivers with a sufficient reputation. Without entering into technicalities, the two main ingredients to that end are, on the one hand, to allow users to express their satisfaction and, on the other hand, to test that a taxi driver is recognized at a sufficient level of satisfaction. For the first task, assuming that only positive marks are taken into account and that the service offered by a taxi driver can be evaluated as good or excellent, the satisfaction of a user can be registered by inserting the tuple `<taxi_driver_id>` once if the evaluation mark is good and twice if it is excellent. Technically, with `taxi_driver_id` the identifier of the taxi driver, this amounts to respectively executing `tell(<taxi_driver_id>)(1)` or `tell(<taxi_driver_id>)(2)`. As regards the second task, imagining that a level of satisfaction 100 is a minimal satisfaction mark for a reasonable driver, making sure that a proposed driver, say identified by `id`, has this quality can be simulated by executing the primitive `ask(<id>(100))`. Note that, as the number of matching tuples is only counted, such a satisfaction level may be reached thanks to the contribution of many users. Of course, different policies should be implemented in the application, for instance to forbid a user to mark a taxi driver more than once a day. It is also worth noting that thanks to the space and time decoupling between information producers and information consumers offered by coordination languages, it is very easy to introduce new users and new taxi drivers in the application. Similarly, distributed concerns can be easily reached by distributing the tuple space following for instance the lines of [17].

As a second example, let us sketch how an intelligent road system can be coded in our proposal. Many big cities suffer from the problem of heavy traffic coming in in the morning and going out in the evening. To tackle such a traffic, it is usual to use some traffic lines in one direction or another depending on the traffic. Building such an application requires to be able to evaluate the density of the traffic in both directions and to suggest to adapt the usage of a traffic line when a certain level of traffic is reached. A pattern similar to the first example can be used for that purpose. Some account should however be given for the density of the traffic typically counted as the number of vehicles entering a zone decreased by the number of vehicle leaving the zone. Technically, the execution of the primitive `tell(v(1))` can simulate a vehicle entering the zone whereas the execution of the primitive `get(v(1))` can simulate its exit from the zone. Observing the certain amount of vehicle, say 500 is reached, can then be detected by executing the primitive `ask(v(500))` in parallel to the tell and get primitives.

Several variations can be built upon this example. For instance, one may distinguish cars, motorcycles, lorries and busses by using distinct tuples, such as respectively `c`, `mc`, `l` and `b`. Similarly, other policies can be envisaged such as requiring the intervention of the police when too many vehicles carrying dangerous goods are detected. One may also transpose this example to similar situations such as the management of big sways in the crowd, in railway stations, underground stations, or even in shops, hospitals or any kind of big public buildings.

The main goal of this paper is however theoretical. By using an abstract language, we shall study the expressiveness of the proposed extension with respect to Linda-like and Gamma-like languages. This is conducted following previous lines of some of the authors: [9,10,19,22–24]. Accordingly, we shall employ De Boer and Palamidessi's modular embedding to analyze the expressiveness of sublanguages and to compare them with sublanguages of Bach as well as sublanguages of multiset rewriting systems. As we shall see, the techniques for establishing the proofs are similar in essence but requires to be adapted to our context of dense tokens.

They allow us to establish that the Dense Bach language presents an increase of expressiveness with respect to the Bach language, due to the capability of the Dense Bach primitives, to manipulate atomically many instances of a same token. As Gamma proposes the same possibility, it is natural in a second step to compare it with Dense Bach. The level of expressiveness stays in the advantage of Gamma, as it is capable to manipulate atomically different tokens, at the contrary of Dense Bach, and also capable to perform atomically different actions, such as telling, asking, getting and nasking. This richness goes thus in the advantage of Gamma, but at the price of a higher level of complexity for the implementation. We thus believe to have identified a language, strictly more expressive than Linda languages, with some capabilities offered by Gamma but, in contrast to the latter, with an implementation complexity similar to Linda.

It is here worth noting that other languages developed by some authors [4,5] have proposed to decorate tuples with an extra field in order to investigate how new properties, essentially related to priority and probability, can be introduced in the Linda coordination model. At the contrary of our approach, by adding such so called "weight" to the tuples, these languages modify them, whereas we only modify the tuple primitives, neither the tuples nor the matching function.

Viroli and Casadei propose in [28] a stochastic extension of the Linda framework, with a notion of tuple concentration, similar to the weight of [4,5] and our notion of density. The syntax of this tuple space is modeled by means of a calculus, with an operational semantics given as a hybrid CTMC/DTMC model. However in their work no expressiveness results are established.

In [15], Gorla studies the relative expressiveness of 16 communication primitives obtained from the combination of four features: synchronous vs asynchronous primitives, monadic vs polyadic data, message passing vs shared dataspaces, and the presence or absence of pattern matching. To that end, the author introduces the notion of reasonable encoding, which