Dynamic Reconfiguration in Coordination Languages

George A. Papadopoulos¹ and Farhad Arbab²

¹ Department of Computer Science, University of Cyprus, 75 Kallipoleos Str.
  P.O.B. 20537, CY-1678 Nicosia, Cyprus
  george@cs.ucy.ac.cy

² Department of Software Engineering, Centre for Mathematics and Computer Science (CWI), Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
  farhad@cwi.nl

Abstract. A rather recent approach in programming parallel and distributed systems is that of coordination models and languages. Coordination programming enjoys a number of advantages such as the ability to express different software architectures and abstract interaction protocols, supporting multilinguality, reusability and programming-in-the-large, etc. In this paper we show the potential of control- or event-driven coordination languages to be used as languages for expressing dynamically reconfigurable software architectures. We argue that control-driven coordination has similar goals and aims with reconfigurable environments and we illustrate how the former can achieve the functionality required by the latter.

Keywords: Coordination Languages and Models; Software Engineering for Distributed and Parallel Systems; Modelling Software Architectures; Dynamic Reconfiguration; Component-Based Systems.

1 Introduction

It has recently been recognized within the Software Engineering community, that when systems are constructed of many components, the organization or architecture of the overall system presents a new set of design problems. It is now widely accepted that an architecture comprises, mainly, two entities: components (which act as the primary units of computation in a system) and connectors (which specify interactions and communication patterns between the components).

Exploiting the full potential of massively parallel systems requires programming models that explicitly deal with the concurrency of cooperation among very large numbers of active entities that comprise a single application. Furthermore, these models should make a clear distinction between individual components and their interaction in the overall software organization. In practice, the concurrent applications of today essentially use a set of ad hoc templates to coordinate the cooperation of active components. This shows the need for proper coordination languages ([2,15]) or software architecture languages ([18]) that can be used to
explicitly describe complex coordination protocols in terms of simple primitives and structuring constructs.

Traditionally, coordination models and languages have evolved around the notion of a **Shared Dataspace**; this is a common area accessible to a number of processes cooperating together towards the achievement of a certain goal, for exchanging data. The first language to introduce such a notion in the Coordination community was Linda with its Tuple Space ([1]), and many related models evolved around similar notions ([2]). We call these models **data-driven**, in the sense that the involved processes can actually examine the nature of the exchanged data and act accordingly.

However, many applications are by nature **event-driven** (rather than data-driven) where software components interact with each other by posting and receiving events, the presence of which triggers some activity (e.g. the invocation of a procedure). Events provide a natural mechanism for system integration and enjoy a number of advantages such as: (i) waiving the need to explicitly name components, (ii) making easier the dynamic addition of components (where the latter simply register their interest in observing some event(s)), (iii) encouraging the complete separation of computation from communication concerns by enforcing a distinction of event-based interaction properties from the implementation of computation components. Event-driven paradigms are natural candidates for designing coordination rather than programming languages; a „programming language based“ approach does not scale up to systems of event-driven components, where interaction between components is complex and computation parts may be written in different programming languages.

Thus, there exists a second class of coordination models and languages, which is control-driven and state transitions are triggered by raising events and observing their presence. A prominent member of this family (and a pioneer model in the area of control-driven coordination) is Manifold ([4]), which will be the primary focus of this paper. Contrary to the case of the data-driven family where coordinators directly handle and examine data values, here processes are treated as black boxes; data handled within a process is of no concern to the environment of the process. Processes communicate with their environment by means of clearly defined interfaces, usually referred to as **input** or **output** **ports**. Producer-consumer relationships are formed by means of setting up **stream** or **channel** connections between output ports of producers and input ports of consumers. By nature, these connections are **point-to-point**, although **limited broadcasting** functionality is usually allowed by forming 1-n relationships between a producer and n consumers and vice versa. Certainly though, this scheme contrasts with the Shared Dataspace approach usually advocated by the coordination languages of the data-driven family. A more detailed description and comparison of these two main families of coordination models and languages can be found in [15].

It has become clear over the last few years that the above mentioned principles and characteristics are directly related to the needs of other similar abstraction models, notably **software architectures** and **configuration** languages such as Conic/Durra ([5]), Darwin/Regis ([9]), PCL ([19]), POLYLITH ([17]), Rapide ([7,10]) and TOOLBUS ([6]). The configuration paradigm also leads naturally to the separation of the component specifying initial and evolving configuration from the actual computational component. Furthermore, there is the need to support reusable (re-) configuration patterns, allow seamless integration of computational components but also substitution of them with others with additional functionality, etc.