Dynamic Reconfiguration in Coordination Languages

George A. Papadopoulos[1] and Farhad Arbab[2]

[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.

In this paper we use the *control-* or *event-driven* coordination language Manifold ([4]) to show how it can be used for developing dynamically evolving configurations of components. The important characteristics of Manifold include compositionality, inherited from the data-flow model, anonymous communication, evolution of coordination frameworks by observing and reacting to events and complete separation of computation from communication/configuration and other concerns. These characteristics lead to clear advantages in large distributed applications.

The rest of the paper is organised as follows: The next section is a brief presentation of the language. The following one illustrates the usefulness of the framework for developing dynamic reconfiguration abstractions. The paper ends with some conclusions, comparison with related work and reference to future activities.

2 The Coordination Language Manifold

Manifold ([4]) is a coordination language which is control- (rather than data-) driven, and is a realisation of a new type of coordination models, namely the Ideal Worker Ideal Manager (IWIM) one ([3]). In Manifold there exist two different types of processes: *managers* (or *coordinators*) and *workers*. A manager is responsible for setting up and taking care of the communication needs of the group of worker processes it controls (non-exclusively). A worker on the other hand is completely unaware of who (if anyone) needs the results it computes or from where it itself receives the data to process. More information on Manifold can be found in [11] and in the paper „Modelling Control Systems in an Event-Driven Coordination Language“ in these proceedings. Note that Manifold has already been implemented on top of PVM and has been successfully ported to a number of platforms including Sun, Silicon Graphics, Linux, and IBM AIX, SP1 and SP2. The language has been used successfully for coordinating parallel and distributed applications ([14,16]), modelling activities in information systems ([13]) and expressing real-time behaviour ([12]).

3 A Case Study in Dynamic Reconfiguration: The Patient Monitoring System

In this section we apply control- or event-driven coordination techniques to model a classical case in dynamic reconfiguration, namely coordinating activities in a patient monitoring system ([19]). In the process, we take the opportunity to introduce further features of Manifold. Due to lack of space, we consider a somewhat simplified version of a real patient monitoring system, but we will hopefully be able to persuade the reader that any additional functionality can be handled equally well by our model.

The basic scenario involves a number of monitors - one for every patient - recording readings of the patient’s health state, and managed by a number of nurses. A nurse can concurrently manage a number of monitors; furthermore, nurses can come and go and thus an original configuration between available nurses and monitors can subsequently change dynamically. A monitor is periodically asked by its supervising nurse to send available readings for the corresponding patient, and it does so. However, a monitor can also on its own send data to the nurse if it notices an abnormal situation. A nurse is responsible for periodically checking the patient's state by asking the corresponding monitor for readings; furthermore, a nurse should also respond to receiving abnormal data readings. Finally, if a nurse wants to leave, s/he notifies a supervisor and waits to receive permission to go; the supervisor will send to the nurse such a permission to leave once all monitors managed by this nurse have been re-allocated to other available nurses. We start with the code for a monitor:

```
Manifold Monitor
{
 port output normal, abnormal.

 stream reconnect BK input    -> *.
 stream reconnect KB normal   -> *.
 stream reconnect KB abnormal -> *.

 event abnormal_readings, normal_readings.
 priority abnormal_readings > normal_readings.

 auto process check_readings is
              AtomicMonitor(abnormal_readings) atomic.

 begin: (guard(input,full, normal_readings),
         terminated(self)).

 abnormal_readings: &self -> abnormal;
                    check_readings -> abnormal;
                    post(begin).
   normal_readings: &self -> normal;
                    check_readings -> normal;
                    post(begin).
}
```

A `Monitor` Manifold comprises three ports: the default `input` port and two output ports, one for sending normal readings and another one for sending abnormal readings. The streams connected to these ports from the responsible nurses have been declared to be `BK` (break-keep) for the input port and `KB` (keep-break) for the two output ports, signifying the fact that in the case of a nurse substitution the data already within the streams to be transmitted to or from the monitor will remain in the corresponding sttream until some other nurse has re-established connection with the monitor. Two local events have been declared, `normal_readings` for the case of handling periodical data readings and `abnormal_readings` for handling the exception of detecting abnormal readings. Note that, for obvious reasons, the priority of the latter has been declared to be higher than that of the former. In the case that both events have been raised (e.g. immediately after a periodic reading an abnormal situation has been detected), the monitor will serve first `abnormal_readings` (if priority had not been specified, the language would have made a non-deterministic choice). Finally, note that `Monitor` collaborates closely with the process `check_readings`, an instance of the predefined *atomic* Manifold `AtomicMonitor`. Atomic Manifolds (and associated processes) are ones written in some other language (typically C or Fortran for the case of the Manifold system). In this case, `AtomicMonitor` can be seen as the device driver for the monitor device.

Initially, Monitor sets a *guard* to its input port, which will post the event normal_readings upon detecting a piece of data in the port. This piece of data is interpreted by monitor as being a periodic request by the responsible for this monitor nurse to get the data readings. It then suspends, by means of calling the predefined primitive terminated(self), and waits for a notification (by means of the corresponding event being posted) by either the guard to send periodic data readings or the process check_readings that some abnormal situation has been detected. Upon detecting the presence of either of the two involved events, Monitor changes to the corresponding state, and sends to the respective output port first its own id (&self) followed by the actual data readings as provided by check_readings. It then loops back to the first (waiting) state, by posting the event begin. It is important to note that Monitor works quite independently from its environment. For instance, it has no knowledge or concern about which nurse (if anyone at all!) is receiving the data it sends. Nor is it affected by any changes in the configuration of the nurses set up. The code for a nurse is shown below:

```
Manifold Nurse
{
 port in normal, abnormal.

 stream reconnect KB normal   -> *.
 stream reconnect KB abnormal -> *.
 stream reconnect BK output   -> *.

 event got_abnormal, got_normal,
       read_data, leave, ok_go.
 priority got_abnormal > got_normal.

 auto process wakeup is WakeUp(read_data,leave).
 auto process process_data is ProcessData.

 begin: (guard(abnormal,full,got_abnormal),
         guard(normal,full,got_normal),
         guard(abnormal,a_disconnected,ok_go),
         ternimated(self)).

 read_data: („SEND_DATA“ -> output, post(begin)).

 got_abnormal: process monitor deref abnormal.
               (monitor.abnormal -> process_data,
                post(begin)).

 got_normal: process monitor deref normal.
             (monitor.normal -> process_data,
              post(begin)).

 leave: (raise(go), post(begin)).

 ok_go: .
}
```

A nurse has two input ports and one output port, a mirror image of how a monitor is defined. It collaborates with two atomic processes: wakeup is responsible for periodically asking the nurse to order a monitor to send data and process_data does the actual processing of data. Furthermore, wakeup also monitors how long a nurse can be on duty. After setting guards into the two input ports (the second guard for abnormal is explained below), nurse suspends waiting for either wakeup to ask her for periodic readings or some monitor to send her abnormal data readings. In the former case, nurse sends a notification to all the monitors that it controls and upon receiving back data it forwards them to process_data. In either of the two cases, a monitor will first send its own id which is then being dereferenced (by means of the deref primitive) to yield a process reference for the monitor in question. This id is then used to connect the monitor's appropriate output port to the input port of process_data so that the readings can be transmitted. This process is being repeated until wakeup lets the nurse know that it can now ask permission to leave. The nurse raises the event go (note here that an event can either be „posted" in which case its presence is known only to the Manifold within which it was posted, or „raised" in which case its presence is known only outside the Manifold within which it was raised), and waits for a notification that it is allowed to leave. Note also that until such a notification has been provided, the nurse is still responsible for its monitors. The requested notification will be provided implicitly by noting that one of the nurse's input ports has now no connections to a monitor. This will be detected because of the presence of the second guard in the abnormal input port with the directive a_disconnected. The nurse can then leave the system.

A number of abstractions have been introduced in the modelling of the nurse, which are of importance to a dynamic configuration paradigm. A nurse is unaware of the number of monitors it supervises. Thus, monitors can be added or deleted from its list of responsibility without affecting the pattern of the nurse's behaviour. Also, the decision on whether a nurse should leave (which can be as simple as noting when a specified time interval has passed or as complicated as taking into consideration additional parameters such as specialization of work, priorities in types of duty, redistribution of workload, etc.), is encapsulated into different components which, as in the case of the number of monitors, they do not affect the basic pattern of behaviour for the nurse. Furthermore, the actual processing of data, which can vary depending on the type of monitor or the patient's case, is also abstracted away. All in all, the *policies* that define the nurses' behaviour have been separated from the actual work to be done, and they can therefore be changed easily, dynamically, and without affecting the interaction patterns between the involved components. The code for the supervisor is as follows:

```
Manifold Supervisor (process setup)
{
 event get_modify.

 begin: (guard(input,full,get_modify),
         terminated(self)).

 go.*nurse: (&nurse -> setup, post(begin)).

 get_modify:process new_nurse deref tuplepick(input,1).
            process mon1 deref tuplepick(input,2).
            process mon2 deref tuplepick(input,3).
            process mon3 deref tuplepick(input,4).
            new_nurse -> (-> mon1, -> mon2, -> mon3),
            mon1.abnormal -> new_nurse.abnormal,
            mon1.normal -> new_nurse.normal
            mon2.abnormal -> new_nurse.abnormal,
            mon2.normal -> new_nurse.normal
            mon3.abnormal -> new_nurse.abnormal,
            mon3.normal -> new_nurse.normal,
            activate(new_nurse), post(begin).
}
```

The `Supervisor` Manifold is responsible for monitoring a number of nurses. It collaborates closely with the atomic process `setup`, which maintains and enforces the policy of the environment with respect to issues such as how many nurses should be active concurrently, how many monitors each nurse should be responsible for, how can the workload of monitor responsibility be distributed evenly to the available nurses, the minimum and maximum amounts of time each nurse should be working before asking to be relieved from duty, etc. Upon receiving a request by some nurse to be allowed to leave, `Supervisor` passes the id of that nurse to `setup` which, among other things, keeps record of which monitors each nurse is responsible for. A new nurse to take over is found and its id along with the ids of the monitors to handle is passed back to `Supervisor`; the latter then sets up the stream connections between the new nurse and the monitors and activates the new nurse (for the sake of brevity here we have assumed a simple scenario where the old nurse is responsible for three monitors, all of which are passed to the new nurse; this of course does not have to be the case). The transferring of the streams connecting the monitors to the new nurse causes the disconnection of the input ports of the old nurse from the whole apparatus. We recall that the nurse has set up a guard process at its input ports which will get activated when it detects such a disconnection; upon the disconnection of its input port `abnormal` from all monitors involved, the old nurse leaves the system and its associated process terminates execution gracefully.

An initial setup with three monitors and two nurses is shown below within the `Main` Manifold which is the first one to commence execution in a Manifold program:

```
Manifold Main()
{
 event go, ok_go.

 auto process n1 is Nurse.
 auto process n2 is Nurse.
 auto process m1 is Monitor.
 auto process m2 is Monitor.
 auto process m3 is Monitor.
 auto process setup is Setup (n1,n2,m1,m2,m3) atomic.
 auto process supervisor is Supervisor(setup).

 begin: (n1-> m1, n2 -> ( -> m2, -> m3),
         m1.normal->n1.normal,m1.abnormal->n1.abnormal,
         m2.normal->n2.normal,m2.abnormal->n2.abnormal,
        m3.normal->n2.normal,m3.abnormal->n2.abnormal).
}
```

In the above piece of code, the instances of the Manifolds forming the initial configuration are first created and activated. In particular, the initial configuration comprises two monitors and three nurses, where the first nurse is responsible for the first monitor and the second one for the rest. A `setup` process (an instance of `Setup`) is also activated and given the previously mentioned five processes as parameters. Finally, a `supervisor` process is also created and given `setup` as parameter. The `begin` state of `Main` sets up the configuration by creating stream connections between the appropriate input-output ports. After that, the apparatus is left to its own devices, evolving dynamically as was described previously during the presentation of the code for `Monitor`, `Nurse` and `Supervisor`.

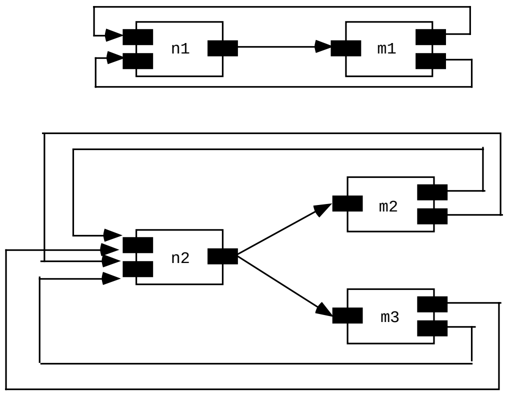

Fig. 1.

We should probably clarify at this point that the actual evolution of the configuration (i.e. the creation of new instances of `Nurse` to substitute other instances) is performed within the `Setup` Manifold, which effectively administers the whole environment, keeps track of changes, etc. We would expect that Manifolds with such a complicated behaviour are atomic processes written, say, in C for convenience and ease of expressiveness. The initial configuration is shown diagramatically below:

4 Conclusions - Related and Further Work

In [8], while describing another configuration mechanism based on I/O abstractions, a number of desirable properties that configuration models should possess are listed. These properties are active and reactive communication, connection-oriented and user-specifiable configuration and support for a variety of communication schemes such as implicit, direct, multi-way, and use of continuous streams. It is worth mentioning here that Manifold supports all these schemes as first class citizens. In addition, Manifold supports complete separation of computation from coordination concerns and a control-driven specification of system transformations, which unlike I/O abstractions, is, in our opinion, more appropriate for configuration programming.

Note that in Manifold, unlike in many other coordination models and languages, a component is oblivious not only to bindings produced by other components but also to whether or not communication is taking place at all or what type of communication this is. This frees the programmer from having to establish when it is the best moment to send and/or receive messages. And of course, the language enjoys the ability for dynamic system reconfiguration without the need to disrupt services or the components having mutual knowledge of structure or location — point-to-point or multicast communications can be configured independently of the computation activity and mapped appropriately onto the underlying architecture.

Furthermore, the stream (or channel) connections that Manifold supports as the basic mechanism for communication between computation components, provide a natural abstraction for supporting continuous datatypes such as audio or video and make this coordination model and its associated language ideal for configuring the activities in, say, distributed multimedia environments. We are currently exploiting this characteristic of Manifold in a recently commenced research project where the language will be used to manage and coordinate, among other activities, the data produced or consumed by media servers.

Finally, Manifold advocates a liberal view of dynamic reconfiguration and system consistency. Consistency in Manifold involves the integrity of the topology of the communication links among the processes in an application, and is independent of the states of the processes themselves. Other languages, such as Conic, limit the dynamic reconfiguration capability of the system by allowing evolution to take place only when the processes involved have reached some sort of a safe state (e.g. quiescence). Manifold does not impose such constraints; rather, by means of a plethora of suitable primitives, it provides programmers the tools to establish their own safety criteria to avoid reaching logically inconsistent states. Furthermore, primitives such as guards, installed on the input and/or output ports of processes, inherently encourage programmers to express their criteria in terms of the externally observable (i.e., input/output) behavior of (computation as well as coordination) processes.

Acknowledgments

This work has been partially supported by the INCO-DC KIT (Keep-in-Touch) program 962144 „Developing Software Engineering Environments for Distributed Information Systems“ financed by the Commission of the European Union.

References

1. S. Ahuja, N. Carriero and D. Gelernter, „Linda and Friends“, *IEEE Computer* **19 (8)**, 1986, pp. 26-34.
2. J. - M. Andreoli, C. Hankin and D. Le Métayer, *Coordination Programming: Mechanisms, Models and Semantics*, World Scientific, 1996.
3. F. Arbab, „The IWIM Model for Coordination of Concurent Activities“, *Coordination'96*, Cesena, Italy, 15-17 April, 1996, LNCS 1061, Springer Verlag, pp. 34-56.
4. F. Arbab, I. Herman and P. Spilling, „An Overview of Manifold and its Implementation“, *Concurrency: Practice and Experience* **5 (1)**, 1993, pp. 23-70.
5. M. R. Barbacci, C. B. Weinstock, D. L. Doubleday, M. J. Gardner and R. W. Lichota, „Durra: A Structure Description Language for Developing Distributed Applications“, *Software Engineering Journal*, IEE, March 1996, pp. 83-94.

6. J. A. Bergstra and P. Klint, „The TOOLBUS Coordination Architecture“, *Coordination'96*, Cesena, Italy, 15-17 April, 1996, LNCS 1061, Springer Verlag, pp. 75-88.
7. C. Chen and J. M. Purtilo, „Configuration-Level Programming of Distributed Applications Using Implicit Invocation“, *IEEE TENCON'94*, Singapore, 22-26 Aug., 1994, IEEE Press, pp. 43-49.
8. K. J. Goldman, B. Swaminathan, T. P. McCartney, M. D. Anderson and R. Sethuraman, „The Programmer's Playground: I/O Abstractions for User-Configurable Distributed Applications“, *IEEE Transactions on Software Engineering* **21 (9)**, 1995, pp. 735-746.
9. J. Kramer, J. Magee and A. Finkelstein, „A Constructive Approach to the Design of Distributed Systems“, *Tenth International Conference on Distributed Computing Systems (ICDCS'90)*, Paris, France, 26 May - 1 June, 1990, IEEE Press, pp. 580-587.
10. D. C. Luckham, „Specification and Analysis of System Architecture Using Rapide“, *IEEE Transactions on Software Engineering* **21 (4)**, 1995, pp. 336-355.
11. Manifold home page, URL: http://www.cwi.nl/~farhad/Manifold.html.
12. G. A. Papadopoulos and F. Arbab, „Coordination of Systems With Real-Time Properties in Manifold“, *Twentieth Annual International Computer Software and Applications Conference (COMPSAC'96)*, Seoul, Korea, 19-23 Aug., 1996, IEEE Press, pp. 50-55.
13. G. A. Papadopoulos and F. Arbab, „Control-Based Coordination of Human and Other Activities in Cooperative Information Systems“, *Coordination'97*, 1-3 Sept., 1997, Berlin, Germany, LNCS 1282, Springer Verlag, pp. 422-425.
14. G. A. Papadopoulos and F. Arbab, „Coordination of Distributed Activities in the IWIM Model“, *International Journal of High Speed Computing*, World Scientific, 1997, Vol. 9 (2), pp. 127-160.
15. G. A. Papadopoulos and F. Arbab, „Coordination Models and Languages“, *Advances in Computers*, Marvin V. Zelkowitz (ed.), Academic Press, Vol. 46, August, 1998, 329-400.
16. G. A. Papadopoulos, „Distributed and Parallel Systems Engineering in Manifold“, *Parallel Computing*, Elsevier Science, special issue on Coordination, 1998, Vol. 24 (7), pp. 1107-1135.
17. J. M. Purtilo, „The POLYLITH Software Bus“, *ACM Transactions on Programming Languages and Systems* **16 (1)**, 1994, pp. 151-174.
18. M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young and G. Zelesnik, „Abstractions for Software Architecture and Tools to Support Them“, *IEEE Transactions on Software Engineering* **21 (4)**, 1995, pp. 314-335.
19. I. Sommerville and G. Dean, „PCL: A Language for Modelling Evolving System Architectures“, *Software Engineering Journal*, IEE, March 1996, pp. 111-121.