ELSEVIER

Future Generation Computer Systems 17 (2001) 1023–1038

FGCS
FUTURE GENERATION COMPUTER SYSTEMS

www.elsevier.nl/locate/future

Configuration and dynamic reconfiguration of components using the coordination paradigm

George A. Papadopoulos [a,*], Farhad Arbab [b]

[a] *Department of Computer Science, University of Cyprus, 75 Kallipoleos Str, P.O. Box 20537, CY-1678 Nicosia, Cyprus*
[b] *Department of Software Engineering, Centre for Mathematics and Computer Science (CWI), Kruislaan 413, 1098 SJ Amsterdam, The Netherlands*

Abstract

One of the most promising approaches in developing component-based (possibly 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, support for multi-linguality, reusability and programming-in-the-large, etc. Configuration programming is another promising approach in developing large scale, component-based systems, with the increasing need for supporting the dynamic evolution of components. In this paper we explore and exploit the relationship between the notions of coordination and (dynamic) configuration and we illustrate 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 the notion of dynamic configuration 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 recognised within the software engineering community, that when systems are constructed of many components, the organisation 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 organisation. 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*

* Corresponding author. Tel.: +357-2-892242; fax: +357-2-339062.
E-mail addresses: george@cs.ucy.ac.cy (G.A. Papadopoulos), farhad@cwi.nl (F. Arbab).

0167-739X/01/$ – see front matter
PII: S0167-739X(01)00043-7

languages [2,24] or *software architecture languages* [28] 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 [5,9,18], 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 [24].

In parallel to the development and evolution of the coordination paradigm, the *configuration* and *dynamic reconfiguration* paradigm has also evolved considerably. Distributed information systems comprise a number of components, both hardware and software (and, in fact, even humans can be viewed as part of this apparatus in approaches such as groupware and collaborative environments). Such a complex and widely dispersed system will need to evolve dynamically due to changes in user requirements, upgrades of software modules, failure or substitution of devices, etc. Furthermore, the components making up the system may be COTS (components-of-the-shelf), legacy systems, etc. adding an extra complexity in managing the dynamic evolution of the whole environment. This has led to the development of *configuration* and *dynamic reconfiguration languages* whose purpose is to provide suitable abstractions for modelling the initial configuration and subsequent dynamic evolution of these complex systems. Some classical examples in the literature of the configuration paradigm are Conic/Durra [6], Darwin/Regis [14], PCL [30], POLYLITH [26], Rapide [10,17] and TOOLBUS [7].

As in the case of coordination, the configuration paradigm also leads naturally to the separation of components specifying initial and evolving configurations from the actual computational components. 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. Thus, there is a valid interest to examine any potential relationship between configuration and coordination, and in particular to what extent we can express the functionality of the former using the latter and vice versa.

In this paper we address the first of the above two goals and we use MANIFOLD 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 for both producers and consumers, 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. In the process of showing how the coordination paradigm can exhibit configuration and dynamic reconfiguration capabilities, we note the natural resemblance that most of the configuration languages have with the control- or event-driven coordination ones. This shows clearly that the latter can also be used for a role traditionally associated with the former.

The rest of the paper is organised as follows. In the next section we present the coordination language MANIFOLD and its underlying model IWIM in sufficient detail so that the programming techniques presented further on can be understood. The following section presents the basic principles of the configuration paradigm and shows its close relationship with the control-driven coordination one as it is expressed by languages such as MANIFOLD. The subsequent main section illustrates the usefulness of the framework for developing dynamic reconfiguration abstractions by means of discussing the implementation of some typical scenarios. The paper ends with some conclusions, comparison with related work and short reference to our future activities.

2. The coordination model IWIM and the language MANIFOLD

MANIFOLD 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]. IWIM is a generic, abstract model of communication that supports the separation of responsibilities and encourages a weak dependence of workers (processes) on their environment. Two major concepts in IWIM are separation of concerns and anonymous communication. Separation of concerns means that computation concerns are isolated from the communication and cooperation concerns into, respectively, worker and manager (or coordinator) modules. Anonymous communication means that the parties (i.e., modules or processes) engaged in communication with each other need not know each other. IWIM-sanctioned communication is either through broadcast of events, or through point-to-point channel connections that, generally, are established between two communicating processes (who do not know each other's identity) by a third party coordinator process. 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. There is no way for a third party process to distinguish a worker from a manager process. This is important because without any special considerations, a manager process can, recursively, manage worker processes which themselves are managers of other workers. At the lowest level of such a hierarchy, are the so-called atomic workers that are in fact computation processes. Computation processes can be written in any conventional programming language. Coordinator processes are clearly distinguished from the others in that they are written in the MANIFOLD language.

MANIFOLD encourages a discipline for the design of concurrent software that results in two separate sets of modules: pure coordination, and pure computation. This separation disentangles the semantics of computation modules from the semantics of the coordination protocols. The coordination modules construct and maintain a dynamic data-flow graph where each node is a process. These modules do no computation, but only change the connections among various processes in the application as prescribed, which changes only the topology of the graph. The computation modules, on the other hand, cannot possibly change the topology of this graph, making both sets of modules easier to verify and more reusable. The concept of reusable pure coordination modules in MANIFOLD is demonstrated, e.g., by using (the object code of) the same MANIFOLD coordinator program that was developed

for a parallel/distributed bucket sort algorithm, to perform function evaluation and numerical optimisation using domain decomposition [4,11].

MANIFOLD possesses the following characteristics:

- *Processes*. A process is a *black box* with well-defined *ports* of connection through which it exchanges *units* of information with the rest of the world. A process can be either a manager (coordinator) process or a worker. A manager process is responsible for setting up and managing the computation performed by a group of workers. Note that worker processes can themselves be managers of subgroups of other processes and that more than one manager can coordinate a worker's activities as a member of different subgroups. The bottom line in this hierarchy is *atomic* processes, which may in fact be written, in any programming language.
- *Ports*. These are named openings in the boundary walls of a process through which units of information are exchanged using standard I/O type primitives analogous to read and write. Without loss of generality, we assume that each port is used for the exchange of information in only one direction: either into (*input* port) or out of (*output* port) a process. We use the notation `p.i` to refer to the port `i` of a process instance `p`.
- *Streams*. These are the means by which interconnections between the ports of processes are realised. A stream connects a (port of a) producer (process) to a (port of a) consumer (process). We write `p.o` → `q.i` to denote a stream connecting the port `o` of a producer process `p` to the port `i` of a consumer process `q`.
- *Events*. Independent of channels, there is also an event mechanism for information exchange. Events are broadcast by their sources in the environment, yielding *event occurrences*. In principle, any process in the environment can pick up a broadcast event; in practice though, usually only a subset of the potential receivers is interested in an event occurrence. We say that these processes are *tuned in* to the sources of the events they receive. We write `e.p` to refer to the event `e` raised by a source `p`.

Activity in a MANIFOLD configuration is *event-driven*. A coordinator process waits to observe an occurrence of some specific event (usually raised by a worker process it coordinates) which triggers it to enter a certain *state* and perform some actions. These actions typically consist of setting up or breaking off connections of ports and channels. It then remains in that state until it observes the occurrence of some other event, which causes the *preemption* of the current state in favour of a new one corresponding to that event. Once an event has been raised, its source generally continues with its activities, while the event occurrence propagates through the environment independently and is observed (if at all) by the other processes according to each observer's own sense of priorities. Fig. 1 shows diagrammatically the infrastructure of a MANIFOLD process.

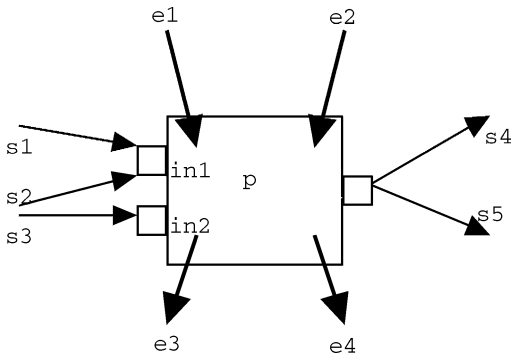

Fig. 1. Control structures of a MANIFOLD process.

As an example, we present the implementation of the dining philosophers problem in MANIFOLD. In this implementation, we model philosophers and forks as separate process types: manifolds `Philosopher` and `Fork`, as shown in the listing, below.

```
1         #define WAIT (preemptall, terminated (self))
2
3         event request, done.
4         manner Eat (process, process, process) import.
5         manner Think (process) import.
6         manner GetTicket () import.
7         manner ReturnTicket () import.
8
9         export Fork()
10        {
11          begin: while true do {
12                   begin: WAIT.
13
14                   request. *phil & *ready. *phil: {
15                     save *.
16                     begin: (raise(ready), WAIT).
17                     done.phil: .
18                     }.
19                 }.
20        }
21
22        export Philosopher ()
23        {
24          event ready.
25
26          begin: while true do {
27                   begin: Think (self);
28                          GetTicket ();
29                          (raise (request, ready), WAIT).
30
31                   ready. *lfork & ready. *rfork: Eat (self, lfork, rfork).
32
33                   end: raise (done);
34                        ReturnTicket ().
35                   }.
36        }
```

Line 1 in the listing defines `WAIT` as a preprocessor macro. What `WAIT` expands into is in fact a common programming idiom that hangs the executing process, waiting for an event from any of its known event sources to cause a state transition.

Line 3 defines `request` and `done` as events in the global scope of this source file. Any module within this source file that uses either of these identifiers without redefining it refers to the corresponding event defined on this line.

Line 4 declares the prototype of a subprogram (manner) called `Eat` that takes three process-type parameters. The keyword `import` states that the body of this subprogram is defined in another source file. In reality, this subprogram

may be a piece of MANIFOLD code in the other source file, or it may be, e.g., a C function. We do not care about the details of `Eat`: whenever a philosopher manages to obtain the two forks it requires to eat, it calls `Eat` to engage in "eating" and passes its own identity plus the identities of its two forks as its parameters (line 31). Similarly, line 5 defines `Think` as another imported subprogram, which is used by a philosopher to do its "thinking" (line 27).

Lines 6 and 7 declare two other imported manners that together implement a "dining ticket" mechanism used to prevent deadlocks (lines 28 and 34). These manners can easily be written in MANIFOLD, but we skip their detail here.

The only way an instance of `Fork` can make a transition out of its `WAIT` state (line 12) is if it observes two event occurrences from the same process, one of which must be an occurrence of the event `request`. The `request` event is defined in this source file and because it does not have the `extern` attribute, this event is not known in any other source file in any application. Within this source file, `request` can be raised only by instances of `Philosopher` (line 29). Thus, the source of the `request` event occurrence (on line 14) can only be an instance of `Philosopher`. The identity of this `Philosopher` instance will be bound to the identifier `phil`, due to the `*phil` construct in the label of this state. This binding restricts the event occurrences that can match the rest of the label: `*ready` can match any event raised by the same source, `phil`, that has also raised `request`. The only thing that can possibly match `*ready` on line 14 is an occurrence of `ready` raised by an instance of `Philosopher` on line 29. Note that there can be two pairs of event occurrences raised by different instances of `Philosopher`, in which case, a `Fork` instance picks one pair non-deterministically.

After transition, a `Fork` instance raises the same `ready` event it has received, and waits for another event (line 16). Although the `ready` event is broadcast by `Fork`, because each philosopher has its own private `ready` event (see below), no one other than the philosopher who raised it and caused a transition in `Fork` to line 14 can react to it. Once a `Fork` instance is in this `WAIT` state (line 16), no event occurrences other than an occurrence of the event `done` from `phil` (i.e., the same instance of `Philosopher` that caused the transition to line 14) can cause it to make a state transition; this is due to the `save` statement on line 15.

To an instance of `Philosopher`, receiving its own private `ready` event means that one of the forks it is potentially entitled to use is now exclusively at its disposal. In return, the `Philosopher` instance must raise the event `done` to inform the forks it has used to "eat" that it is done with them. Thus, a committed `Fork` ends its wait in each iteration (line 17) when it receives an occurrence of `done` raised by the same philosopher it had committed itself to on line 14. At this point, the next iteration starts.

Because the event `ready` is declared inside the body of `Philosopher` (line 24), every instance of `Philosopher` will have its own unique, private `ready` event. Analogous to `Fork`, an instance of `Philosopher` enters an infinite loop upon its activation (lines 26–35). In each iteration of this loop, a `Philosopher` instance first does its thinking (line 27), then waits to obtain a dining ticket (line 28), and finally, declares its intention to eat by raising the events `request` and its private `ready`, and goes into a wait state. If and when two instances of `Fork` declare their exclusive commitment to this `Philosopher` instance, it will have two occurrences of its own `ready` event raised by them in its event memory. This allows it to make a transition to the state on line 31, where the identities of the two forks will be bound to `lfork` and `rfork`. Now the `Philosopher` instance eats, and once done, it raises the event `done` to release its committed forks (line 33), returns its dining ticket (line 34), and goes on to its next iteration.

There will be 11 processes running in this application: an instance of `Main`, five instances of `Philosopher`, and five instances of `Fork`. Each of these 11 processes will actually be a light-weight process (preemptively scheduled thread) in some task instance (heavy-weight process). The actual number of task instances that house these processes can range from 1 to 11, and they can run on the same actual (single or multi-processor) host or on several (homogeneous or heterogeneous) such hosts over a network. None of this detail is relevant at the level of the source code, and the same compiler produced object code can be linked with different specifications in the **mlink** and **config** input to tailor the desired run-time configuration.

The `Main` program, shown below, simply creates and activates five instances of `Philosopher` and five instances of `Fork`, and arranges them in a circular configuration around the virtual table. `Main` accomplishes these

introductions by making each instance of philosopher and fork sensitive to the events raised by the other processes it must know.

```
Main()
{
  auto process phil1 is Philosopher ().
  auto process phil2 is Philosopher ().
  auto process phil3 is Philosopher ().
  auto process phil4 is Philosopher ().
  auto process phil5 is Philosopher ().
  auto process fork1 is Fork ().
  auto process fork2 is Fork ().
  auto process fork3 is Fork ().
  auto process fork4 is Fork ().
  auto process fork5 is Fork ().
  begin: ((phil1->,phil2->)->fork1,
          (phil2->,phil3->)->fork2,
          (phil3->,phil4->)->fork3,
          (phil4->,phil5->)->fork4,
          (phil5->,phil1->)->fork5).
  end: .
}
```

More information on MANIFOLD can be found in [18]; 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, among others, coordinating parallel and distributed applications [4,11,23,25], modelling activities in information systems [22] and expressing real-time behaviour [15,21].

3. Basic concepts of the configuration paradigm and relationship with coordination

The basic principles of configuration programming have been summarised in [13] as follows: (i) The configuration language used for structural description should be separate from the programming language used for basic component programming. (ii) Components should be defined as context independent types with well-defined interfaces. (iii) Using the configuration language, complex components should be definable as a composition of instances of component types. (iv) Change should be expressed at the configuration level, as changes of the component instances and/or their interconnections.

Furthermore, [8] outlines the responsibilities of configuration languages which are: (i) grouping of processes into components; (ii) parameterising and instantiating components; (iii) establishing configuration of a set of components as communication between them; (iv) expressing constraints that specify when configurations and any evolutions of them are permissible. Finally, in [12], 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.

An even superficial examination of some of the most well-known configuration languages [6,10,12,14,16,17,26, 29,30] reveals that most of them are based on some common notions, namely: (i) Components, as the basic building blocks for process descriptions. More often than not components in configuration languages are treated as black boxes whose internals are of no interest at the configuration level. (ii) Ports, as the interaction mechanism of

components with their environment and realised as well-defined "openings" (input or output) on the boundaries of components. (iii) Connectors, as the mechanism for communication between components and realised by means of channels or streams connecting output ports to input ports and vice versa. The above features are coupled with some mechanism for establishing reconfiguration scenarios and when it is safe for some configuration to evolve to another one (for more detailed discussions on these topics see, among others, [8,13,14,16,24,29]).

It is therefore clear from the above discussion that the configuration paradigm has many common features with the control-driven coordination one. And, furthermore, those coordination languages such as MANIFOLD have the potential to also play the role of configuration ones. In the next section we further elaborate on this approach by showing how MANIFOLD can express some typical configuration scenarios.

4. Configuration and dynamic reconfiguration in MANIFOLD

In this main section of the paper we show how control- or event-driven coordination languages, such as MANIFOLD, can express configuration and dynamic reconfiguration properties, by modelling two classical scenarios. Our first case study is the set up and evolution of a community of distributed components associated with each other by means of forming a ring. The second one is the well-known patient monitoring system.

4.1. Principles of configuration in distributed systems: evolution of a token ring

The scenario, which was initially presented in [19], is as follows: a set of components accesses a common resource, each at a time. The components are connected in a cyclic manner, with a token continuously visiting each component in turn (for a specified period of time). While a component has hold of the token, it can access the common resource. Furthermore, it should be possible for this initial set up to evolve, in the sense that new components may dynamically enter the cycle or existing ones removed from it. Such evolutions should be done with a minimum of disruption in the activities of the configuration.

Our solution to this scenario is presented below, followed by detailed explanation:

```
event join, joined, join_broken, break.
manner connect2 (process p1, p2)
{ begin: p1->p2. }
manner connect3 (process p1, p2, p3)
{ begin: p1->p2->p3. }.
manifold Component (manifold AccessResource (event))
{
port in time_slice, pid.
event end_access, token_arrived, ok_join, begin, release, leave.
priority leave<end_access.
variable prev, next, token, coordinator.
process shared_resource is AccessResource (leave).
/* Join the token ring */
begin: (raise (join), guard (input, full, ok_join), WAIT).
ok_join: { process p1 deref tuplepick (input,1).
           process p2 deref tuplepick (input,2).
           begin: (connect (p1,self,p2), raise (joined), prev=p1, next=p2,
                   guard (input, full, token_arrived)).
           begin1: WAIT.
/* get token and access shared resource */
```

```
          token_arrived: token=input;
                        (if (token==``server''
                          then (activate (shared_resource),
                               alarm (time_slice,end_access),
                               post (begin1)),
/* become responsible for adding to the ring new components */
                        (if (token==``coordinator''
                        then (coordinator=true, alarm(random,release),
                               post(begin1)).
/* release shared resource and forward token to the next process */
          end_access|end.shared_resource: (deactivate (shared_resource),
                                           ``server->next''
                                          post (begin1)).
/* pass responsibility of adding new processes to the next process */
          release: (``coordinator->next'', coordinator=false,
                    post (begin1)).
/* adding a new process in the ring */
          join.*p: if (coordinator==true)
                   then { save *
                          begin: ((<<&self,next»->p, WAIT.
                          joined. p: .
                          };
                          post (begin1).
                   else post (begin1).
/* leave a configuration */
          leave: raise (join_broken),
                 if (coordinator==true)
                 then (``coordinator->next, coordinator=false,
                 &next->prev.pid,&prev->next.id, WAIT).
          break.prev: halt.
/* re-arrange links after a process has left the configuration */
          join_broken.next: { process p deref pid.
                              begin: (connect2 (self, p), raise (break),
                                       post (begin1)). }.
          join_broken.prev: prev=pid.
        }
}
```

The general idea of the solution we propose is as follows. Every component already a member of the configuration is aware of the identity of the two components with which it is connected in a circular order. Two tokens travel through this apparatus, visiting in turn each component in the ring: the first, allows the component which has it to access the shared resource for as long as it holds the token; the second, effectively elevates the component which has it to the level of being, temporarily, the coordinator responsible for adding new components in the apparatus. The first token is held by a component for a pre-specified period of time (time slice) whereas the second one remains with a component for some random period of time before moving on to the next one. In that respect, new components are added in a random way around the ring and the overhead of modifying the existing connections distributes over all the existing components. However, the responsibility of a component removing itself from the ring remains with the component itself.

More to the point, at the beginning a component wishing to join the token ring raises the event `join` and then suspends waiting to receive in its default input port a pair of process ids between which it will insert itself. This is achieved by inserting a guard in its input port which will generate the event `ok_join` once the pair of process ids has arrived there. The event `join` is observed by all processes in the ring but only the one currently being the coordinator will react to it. In particular, this process will have already received the token "`coordinator`" and will have set its boolean variable `coordinator` to `true`. The coordinator will then send the pair (`self, next`) with the process ids of itself and the next process in the ring to the component wishing to join the set up. It then suspends and will not perform any other operation (including removing itself from the ring) until it receives from the first process the event `joined` confirming that that process has indeed joined in.

Once the process `P` wishing to join the ring has received the pair of process ids `P1` and `P2`, it spawns the manner (i.e. parameterised manifold) `connect` which sets up the new links by creating stream connections between the output port of `P1` and the input port of `P` on the one hand, and between the output port of `P` and the input port of `P2` on the other hand. It then raises the event `joined` letting everyone know that it is now a member of the token ring.

Once a member of the token ring, a component sets another guard in its default input port waiting to receive either of the two tokens. If it receives the token "`server`", it accesses the shared resource by activating the atomic process `shared_resource`, an instance of the manifold `Access_Resource`. The process `shared_resource` will keep accessing the shared resource for the time interval indicated in the value passed to its input port `time_slice`. Once this time interval has elapsed the primitive `alarm` will raise the event `end_access` which will cause the termination of the process `shared_resource` and the forwarding of the token `server` to the next process in the ring.

If a component receives the token "`coordinator`" then for a random period of time it is responsible for handling requests by external to the ring processes to enter it and become members of it. This is done as described above. Once this random period of time has elapsed, the primitive `alarm` will raise the event `release` which will cause the token `coordinator` to be forwarded to the next process in the ring, passing along also the responsibilities of handling further requests for new memberships to the ring.

Finally, if the agent `shared_resource` has no further need of accessing the shared resource, it generates the event `leave`. This causes the manifold controlling `shared_resource` to raise the message `join_broken` in order to inform the affected components that it is about to leave the ring. The affected components of a component `P` that want to leave are the ones on its left and right, say `P1` and `P2`, with which it holds stream connections. `P` will send to `P1` the process id of `P2`, will also send to `P2` the process id of `P1` and will then suspend waiting for the event `break`. If a process `P1` detects the raising of `join_broken` from its immediate forward neighbour (whose id is always stored in the variable `next`) it then connects its own output port (which until now is connected to the input port of `P`) to the input port of `P2` and raises the event `break`. If a process `P2` detects the raising of `join_broken` from its immediate previous neighbour (whose id is always stored in the variable `prev`) it simply updates the `prev` variable to point now to the new immediate previous process (whose id has been provided by `P` via the `pid` input port). Once the process `P` detects the presence of the event `break`, it knows that it can safely halt. Note that if the component that wants to leave the ring is the coordinator, the token "`coordinator`" is passed immediately to the next process. Similar problems do not arise with the case of the token "`server`" because this is controlled by the same process (`shared_resource`) which will decide when to leave.

Note that the configuration and reconfiguration aspects of the scenario are separated from the computational concerns of the application, i.e. what exactly `shared_resource` does or what short of shared resource is being accessed. Furthermore, the code itself of `shared_resource` may be substituted by another computational module without affecting the apparatus. The inclusion of a new member into the ring is a mostly distributed activity with all processes sharing from time to time this responsibility of house keeping (more sophisticated and fair scenarios are also possible to have). The removal of a member is a purely distributed process affecting only the three components involved in it. By virtue of MANIFOLD's event system and state transition semantics, any race conditions that may arise due to a process being at the same time a coordinator but also wanting to leave the system are handled by the system and the declared priority on handling events (with `leave` having the lowest priority).

4.2. 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 [30]. 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 our presentation 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), WAIT).
 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 channels 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 channels to be transmitted to or from the monitor will remain in the corresponding channel 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 nurse (responsible for this monitor) to get the data readings. It then suspends 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 neither has 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), WAIT).
 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 here 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 specialisation 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. We end our example with the code for the supervisor:

```
manifold Supervisor
{
 event get_modify.
 begin: (guard (input,full,get_modify), terminated (self)).
 go.*nurse: &nurse->set up, 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.

As we have highlighted above, the policies of the system regarding the functionality of its components have been clearly separated from configuration concerns. In addition to dynamically adding and deleting components or changing their inter-communication patterns, we are therefore also able to dynamically modify their functionality — in particular, upgrade it or enhance it. For instance, assume that at some stage it may be necessary to enhance the system with a data logger which preserves the periodic readings of the patients' state in order to create a medical

history database. This requires the actual introduction of the data logger process into the system, which we assume that it has to be part of the functionality of process_data; thus the latter instance must be substituted with a higher version. The code below shows how the Nurse manifold can be modified to be able to accomplish such an upgrade:

```
manner handle_data (port out monitor, process data_pr)
{
 begin: monitor->date_pr.
}
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.
 auto process pr_data is variable.
 pr_data=process_data..
 begin: (guard (abnormal,full,got_abnormal), guard (normal,full,got_normal),
         guard (abnormal,a_disconnected,ok_go), WAIT).
 read_data: ''SEND_DATA'' → output,post(begin)).
 got_abnormal: process monitor deref abnormal.
               (handle_data (monitor.abnormal,pr_data), post(begin)).
 got_normal: process monitor deref normal.
             (handle_data (monitor.normal,pr_data), post(begin)).
 upgrade.*system: (system->pr_data, post(begin)).
 leave: (raise (go), post (begin)).
 ok_go: .
}
```

In the above modified version of Nurse we have introduced the use of the manner handle_data (we recall that a manner is a parameterised manifold) which effectively abstracts away the detail of which particular process is used to handle the data sent by a monitor. Within Nurse we have introduced a variable pr_data which at any time holds a reference to the current version of the program processing the data. We have also introduced a new state, which will be activated upon Nurse observing the raising of the event upgrade by some other process. Nurse will then connect to the output port of that process, get a process id for the new version of the data handling process, and update the variable pr_data. Thus, every time the manner handle_data is called, will forward the data from the monitor to the most current version of the data handling process (we assume here that by using techniques similar to the ones illustrated above, process_data and any other similar process can detect that it is not needed anymore and terminate itself gracefully). This updating can of course be done as often as it is needed.

5. Conclusions — related and further work

In this paper we have examined the potential for control- or event-driven coordination languages to address the needs of the configuration paradigm. We have shown that configuration and dynamic reconfiguration languages are

based on features, which are quite similar with the family of coordination languages in question. We have further demonstrated the ability of a typical member of the control-driven coordination languages, namely MANIFOLD, to act as configuration language.

In [27] there is an interesting comparison between configuration and coordination. That paper claims that these two concepts are central in the design and implementation of middleware systems; it further defines configuration as the establishment of the structure of a system and coordination as the interaction between the components forming the system. We believe we have shown that languages such as MANIFOLD can naturally integrate both these aspects into the same language formalism.

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 multi-cast 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 multi-media environments [15,20]. 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 [13,14,30] 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). Yet other models rely on the support of atomic transactions by the underlying system to guarantee that dynamic evolutions will not reach inconsistent states [29]. 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) behaviour of (computation as well as coordination) processes.

Acknowledgements

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, D. Gelernter, Linda and friends, IEEE Comput. 19 (8) (1986) 26–34.
[2] J.-M. Andreoli, C. Hankin, D. Le Métayer, Coordination Programming: Mechanisms, Models and Semantics, World Scientific, Singapore, 1996.
[3] F. Arbab, The IWIM model for coordination of concurrent activities, in: Proceedings of the First International Conference on Coordination Models, Languages and Applications (Coordination’96), Cesena, Italy, April 15–17, 1996, Lecture Notes in Computer Science, Vol. 1061, Springer, Berlin, pp. 34–56.
[4] F. Arbab, C.L. Blom, F.J. Burger, C.T.H. Everaars, Reusable coordinator modules for massively concurrent applications, Softw.: Pract. Experience 28 (7) (1998) 703–735.

[5] F. Arbab, I. Herman, P. Spilling, An overview of MANIFOLD and its implementation, Concurrency: Pract. Experience 5 (1) (1993) 23–70.
[6] M.R. Barbacci, C.B. Weinstock, D.L. Doubleday, M.J. Gardner, R.W. Lichota, Durra: a structure description language for developing distributed applications, Softw. Eng. J. IEE 8 (2) (1993) 83–94.
[7] J.A. Bergstra, P. Klint, The TOOLBUS coordination architecture, in: Proceedings of the First International Conference on Coordination Models, Languages and Applications (Coordination'96), Cesena, Italy, April 15–17, 1996, Lecture Notes in Computer Science, Vol. 1061, Springer, Berlin, pp. 75–88.
[8] J.M. Bishop, Languages for configuration programming: a comparison, UP–CS Technical Report, 1994.
[9] M.M. Bonsangue, F. Arbab, J.W. de Bakker, J.J.M.M. Rutten, A. Scutella, G. Zavattaro, A transition system semantics for the control-driven coordination language MANIFOLD, Theoret. Comput. Sci. 240 (1) (2000) 3–47.
[10] C. Chen, J.M. Purtilo, Configuration-level programming of distributed applications using implicit invocation, IEEE TENCON'94, Singapore, August 22–26, 1994, IEEE Press, New York, pp. 43–49.
[11] K. Everaars, F. Arbab, B. Koren, Dynamic process composition and communication patterns in irregularly structured applications, Concurrency: Pract. Experience 12 (2–3) (2000) 157–174.
[12] K.J. Goldman, B. Swaminathan, T.P. McCartney, M.D. Anderson, R. Sethuraman, The programmer's playground: I/O abstractions for user-configurable distributed applications, IEEE Trans. Softw. Eng. 21 (9) (1995) 735–746.
[13] J. Kramer, Configuration programming — a framework for the development of distributed systems, in: Proceedings of the IEEE International Conference on Computer Systems and Software Engineering (COMPEURO'90), Israel, May 1990, IEEE Press, New York.
[14] J. Kramer, J. Magee, A. Finkelstein, A constructive approach to the design of distributed systems, in: Proceedings of the 10th International Conference on Distributed Computing Systems (ICDCS'90), Paris, France, May 26–June 1, 1990, IEEE Press, New York, pp. 580–587.
[15] T.A. Lemniotes, G.A. Papadopoulos, Real-time coordination in distributed multimedia systems, in: Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS 2000), Eighth International Workshop on Parallel and Distributed Real-time Systems (WPDRTS 2000), Cancun, Mexico, May 1–2, 2000, Lecture Notes in Computer Science, Vol. 1800, Springer, Berlin, pp. 685–691.
[16] O. Loques, A. Sztajnberg, J. Leite, M. Lobosco, On the integration of configuration and meta-level programming approaches, Reflection and Software Engineering, Lecture Notes in Computer Science, Vol. 1826, Springer, Berlin, 2000, pp. 191–210.
[17] D.C. Luckham, Specification and analysis of system architecture using Rapide, IEEE Trans. Softw. Eng. 21 (4) (1995) 336–355.
[18] MANIFOLD home page. URL: http://www.cwi.nl/~farhad/manifold.html.
[19] N.H. Minsky, V. Ungureanu, W. Wang, J. Zhang, Building reconfiguration primitives into the law of a system, in: Proceedings of the Fourth International Conference on Configurable Distributed Systems, MD, USA, May 6–8, 1998, IEEE Press, New York.
[20] S. Mitchell, H. Naguib, G. Coulouris, T. Kindberg, Dynamically reconfigurable multimedia components: a model-based approach, in: Proceedings of the Eighth ACM SIGOPS European Workshop, Sintra, Portugal, September 7–10, 1998, ACM, New York, pp. 40–47.
[21] G.A. Papadopoulos, F. Arbab, Coordination of systems with real-time properties in MANIFOLD, in: Proceedings of the 20th Annual International Computer Software and Applications Conference (COMPSAC'96), Seoul, Korea, August 19–23, 1996, IEEE Press, New York, pp. 50–55.
[22] G.A. Papadopoulos, F. Arbab, Control-based coordination of human and other activities in cooperative information systems, in: Proceedings of the Second International Conference on Coordination Models and Languages, September 1–3, 1997, Berlin, Lecture Notes in Computer Science, Vol. 1282, Springer, Berlin, pp. 422–425.
[23] G.A. Papadopoulos, F. Arbab, Coordination of distributed activities in the IWIM model, Int. J. High Speed Comput. 9 (2) (1997) 127–160.
[24] G.A. Papadopoulos, F. Arbab, Coordination models and languages, in: M.V. Zelkowitz (Ed.), Advances in Computers, Vol. 46, Academic Press, New York, August 1998, pp. 329–400.
[25] G.A. Papadopoulos, Distributed and Parallel Systems Engineering in MANIFOLD Parallel Computing (special issue on Coordination), Vol. 24, No. 7, Elsevier, Amsterdam, 1998, pp. 1107–1135.
[26] J.M. Purtilo, The POLYLITH software bus, ACM Trans. Programming Languages Syst. 16 (1) (1994) 151–174.
[27] M. Radestock, S. Eisenbach, Component coordination in middleware systems, in: Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), Lake District, UK, September 15–18, 1998, Springer, Berlin, pp. 225–240.
[28] M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, G. Zelesnik, Abstractions for software architecture and tools to support them, IEEE Trans. Softw. Eng. 21 (4) (1995) 314–335.
[29] S.K. Shrivastava, S.M. Wheater, Architectural support for dynamic reconfiguration of large scale distributed applications, in: Proceedings of the Fourth International Conference on Configurable Distributed Systems (CDS'98), MD, USA, May 4–6, 1998, IEEE Press, New York.
[30] I. Sommerville, G. Dean, PCL: a language for modelling evolving system architectures, Softw. Eng. J. IEE 11 (2) (1996) 111–121.

George A. Papadopoulos is an Associate Professor in the Department of Computer Science at the University of Cyprus in Nicosia, Cyprus. His current research interests are in the areas of component-based software engineering, parallel and distributed systems, programming languages and cooperative information systems in which he has over 50 publications. He is a recipient of the 1995 ERCIM Human Capital Mobility award. More information about him can be found at his personal web page http://www.cs.ucy.ac.cy/~george.

Farhad Arbab is a Senior Researcher in the Department of Software Engineering at CWI, the Netherlands. His research areas are software engineering, parallel and distributed systems and programming languages.