Modelling Control Systems in an Event-Driven Coordination Language

Theophilos A. Limniotes and George A. Papadopoulos

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

Abstract. The paper presents the implementation of a railway control system, as a means of assessing the potential of coordination languages to be used for modelling software architectures for complex control systems using a components-based approach. Moreover, with this case study we assess and understand the issues of real time, fault tolerance, scalability, extensibility, distributed execution and adaptive behaviour, while modelling software architectures. We concentrate our study on the so-called control- or event-driven coordination languages, and more to the point we use the language Manifold. In the process, we develop a methodology for modelling software architectures within the framework of control-oriented coordination languages.
Keywords. Concepts and languages for high-level parallel programming; Distributed component-based systems; Software Engineering principles; High-level programming environments for Distributed Systems.

1 Introduction

A number of programming models and associated software development environments for parallel and distributed systems have been proposed, ranging from ones providing elementary parallel constructs (such as PVM and MPI) to ones offering higher level logical abstractions such as skeletons, virtual shared memory metaphors (such as Linda), coordination models ([4]), software architectures ([5]), etc. It would be interested to examine the potential of those models in modelling real-life non-trivial applications and in the process develop a software engineering methodology for their use.

In this paper we present some of the main components of implementing a railway control system using the coordination language Manifold ([1]). It is a typical real world problem which, apart from its operational aspects, it necessitates the addressing of several other requirements that relate to *real-time*, *fault tolerance* and *adaptive behaviour*. In the process, we present a methodology for developing such real-life applications using the control-driven coordination metaphor.

The rest of the paper is organised as follows: the next section describes briefly the case study while the following one is a brief introduction to the coordination language

M. Bubak et al. (Eds.): HPCN 2000, LNCS 1823, pp. 177-186, 2000.
© Springer-Verlag Berlin Heidelberg 2000

Manifold. The description of the implementation of the case study in Manifold is then presented with some concluding remarks at the end of the paper.

2 Description of the Case Study

The scope of designing a control system is to process raw data obtained from the environment through sensing devices and gauges, determine the model parameters that describe the environment, decide interdependencies of change of states, adapt problem solving routines, and provide control information to the users. Furthermore, other requirements to be met include *functionality, timeliness, fault-tolerance, degraded modes, extensibility,* and *distribution.*

In particular, we study the modelling of a railway system ([2]) consisting of railway tracks, junctions, and platforms. A number of trains is expected to be travelling across the network. The system should be able to monitor the position of each of the trains, access the current situation of the trains, be able to predict and cope with future developments, and take into account timetables (and follow the specified schedules as much as possible). Moreover, speed variation should be avoided whenever possible (speed adjustments) and direction adjustments should be supported. Other desirable features should include *scalability* and *extensibility* (e.g. modifying the network topology) and *fault tolerance* (e.g. coping with train failures).

3 Manifold

Manifold ([1]) is a control-driven coordination language. In Manifold there are 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. Manifold possess 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.
- *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 streams, 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. Figure 1 below shows diagramatically the infrastructure of a Manifold process.

The process `p` has two input ports (`in1`, `in2`) and an output one (`out`). Two input streams (`s1`, `s2`) are connected to `in1` and another one (`s3`) to `in2` delivering input data to `p`. Furthermore, `p` itself produces data which via the `out` port are replicated to all outgoing streams (`s4`, `s5`). Finally, `p` observes the occurrence of the events `e1` and `e2` while it can itself raise the events `e3` and `e4`. Note that `p` need not know anything else about the environment within which it functions (i.e. who is sending it data, to whom it itself sends data, etc.).

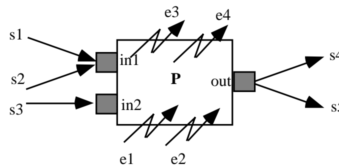

Fig. 1.

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. For more information on the language and its potential uses we refer the interested reader to [1] and [5].

4 Implementing the Railway System as Components

Our aim is to construct a system that consists of static and dynamic components. The static ones are those that comprise a railway network, namely platforms, tracks, and junctions; we refer to them as rail components. These can be altered in the sense that entities of this kind can be added or removed, but these operations are not continual

and time dependent as it happens with the dynamic components. The number and variation of both types of components directly relates to the scalability of the system.

The dynamic components are the trains themselves. There is a potential number of trains remaining dormant, that can be activated at any moment. Their 'course' or 'route' can be seen as a collection of static objects that the train's trip comprises. Other dimensions of continual change are the location of the train that is time dependent and the status of the train that is indirectly related to the status of the 'next one to visit' rail component. The two entities above can trigger at any time the redefinition of part of the route, including the destination platform of the train.

The railway system model built in Manifold consists of coordinating modules at a higher level and computational components at a lower level. In this particular application the computational components are written in C. Due to lack of space, we refrain from describing these modules in this paper. Irrespectively of its functionality, each module represents a building block in the architecture of the system. At one level the coordination components specify a number of transition pipelines that constitute the deportment of a process. At a higher level another transition system defines the interaction among such processes. Thus the pattern of the system formed can be described as transitions (at a higher level) to predefined algorithms (at a lower level). It should be made clear at this stage that the first level can have processes written in both computational languages which have their external behaviour observable, and processes written in Manifold defining different sets of transitions into the process structure.

Here follows a description of the coordination component. We should emphasize at this point that only the most important parts of the code are shown below in the presented manifolds. The whole application comprises around 2,000 lines of Manifold and C code. Thus, the presented functionality of the major components described below is somewhat simplified. Note that the names within the boxes refer to atomic (C code) processes, which are called by the respective manifolds and perform some lower level (namely irrelevant to the coordination protocols used) function.

Train Coordinator Code. An instance of this Manifold receives at a port the identification number of a train that is about to depart. For the current model this instance has to be predefined in `Main,` so that the particular train's id can be associated with a particular number of already activated `Train` instances that can run in parallel.

Let such a `Train` instance be `train1`. This `train1` will have to observe events that are only broadcast by particular `Platforms` and `Junctions` (which the train in question will go through during its trip) to which `train1` sends requests for passing. A particular input port (say `train1.rail`) is assigned for this job. This port then is passed as a parameter to the manner `CheckInput` whose function is to make any process dereferenced from this input port an observable source of events.

Similarly, provided that we have such a reference of a `Train` instance `(&train1` that is), connected to the input port of another process instance which in turn is also a parameter for `CheckInput`'s input port (say `platfrom1.train`), `train1` will be able to have its events *observable* by that other process instance. Moreover it has to be sorted out through which port each event of another process will be made

available to train1 This can be done by dereferencing a local event to another input port (say train1.get) which has to be connected to the output port of the event it expects (say platform1.send). An example of such code follows:

```
Train ()
port in rail
port in get1, get2, get3./* get three different events */
port in depart    /* get train id from Actual Rail info*/
{
 event get_thru deref get1.
 event get_thru_next deref get2.
 event gone deref get3.
 process i is variable(1).
 begin:
       CheckInput(rail);
       while true do {
       begin:
            /* steps of going through a rail component */
            if (i==1) then (raise(get_thru),WAIT);
            if (i==2) then (raise(get_thru_next),WAIT);
            if (i==3) then (raise(gone),WAIT);
            if (i==4) then (post(end)).
            /*reply to events above done */
            get_thru | get_thru_next | gone:i=i+1.
       end:. }
}
```

Platform Coordination Code. The instances of this Manifold are activated at the beginning of Main() as 'auto' instances. Each platform instance is given its id as a parameter at the declaration statement. In that way, each platform instance is associated with a particular platform right from the beginning.

Each Platform instance has to observe events which, in fact, are requests from Train instances. A particular input port is assigned for this purpose. This port then is passed as a parameter to an input port (say platform1.train) of the manner CheckInput, so that any instance of a process dereferenced from that input port, is an observable source of events.

A Platform instance can 'reply' to events of a Train instance, which in this case can be consider as requests, when its reference, i.e. &platform1 is connected to the input port of that train which in turn is also a parameter for CheckInput's port. The platform code follows below.

```
Platform(Manifold Get_Block_Status(event),
         Manifold Get_Block_Free, port in id)
port in  train, next_id.
port out send1, send2, send3.
{
 event OK_free, OK_free_next, gone_OK, replies.
 event replies.
 auto process ID is variable.
 auto process next_ID is variable.
 begin: heckInput(train);
```

```
          (id->ID,next_id->next_ID,post(replies)).
   replies: while true do {
            begin:(&OK_free->send1,
                   &OK_free_next->send2,&gone_OK->send3).
            OK_free.*train:{ event im_free.
                             auto process gs is
                               Get_Block_Status(im_free).
                             begin:Trip(train);
                                   ID->(->gs,->ID).
                             im_free.gs:raise(OK_free).
                      end:.}.
            OK_free_next.*train:{ event im_free.
                                  auto process gs is
                                Get_Block_Status(im_free).
                                  begin:Trip(train);
                                next_ID->(->gs,->next_ID).
                     im_free.gs:{ auto process gf is
                                          Get_Block_Free.
                                   ID->gf;
                    (raise(OK_free_next),raise(OK_free)).
                                 }. end:.}.
            gone_OK.*train:{ auto process gf is
                                          Get_Block_Free.
                             begin: next_ID-> gf;
                                    Trip(train);

  (raise(gone),raise(OK_free_next),raise(OK_free)).
  } } }.
```

Junction Coordination Code. The operation of a `Junction` instance is basically the same as that of a platform, and the id of the next track that the train will pass is provided by

```
train.next_track->junction.next_id.
```

as well as

```
train.next_track2->junction.next_id2
```

which is an alternative choice for a diversion, in case that the first track is occupied. The availability of a second choice in its route is decided by the train instance itself. The `junctions` instance role in this is that it checks serially for this second choice (if it is free) if and only if checking the first choice has not preempted the control to the '`im_free`' state.

Assembling All Components Together. The above described major components are assembled together as indicated by the figure below. The following description is concerned with the externally observable behaviour, and as all but one atomic are activated from within the coordinator manifolds, we are not concerned with these atomics at this level.

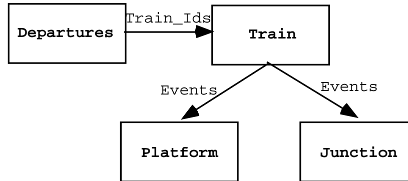

Fig. 2.

To show a particular scenario, we consider a simple case (effectively a snapshot of a more general scenario) of 3 trains travelling on a railway configuration involving a junction and a platform. There are two tracks connected to the junction and one tracj leaving the junction and ending at the platform. Two of the trains are approaching the junction travelling on either track, whereas the third one is stationed at the platform. In `Main` there are two streams created that connect two output ports of a `Departure` atomic with two input ports of two other instances of the `Train` coordinator. All instances of `Platform` and `Junction` controllers can accept requests. So, there are two `Train` instances running in parallel with this model, sending out requests to the already activated `Platform` and `Junction` instances. For the requests and replies there should be streams hardwired between pairs of `'send'` and `'get'` ports of particular `Platform/Junction` and `Train` instances respectively. Also, a guard is required to be placed at `Train`'s '`end_port`' in order to be able to preempt to a state that defines a new transition configuration. Finally, we have to hardwire the reference of each instance of a process with the dereferenced port of the instance that it has to send an event to.

A number of `Train` instances have to be activated at the `begin` state of `Main`, together with an instance `Departures` atomic. The `Departures` is a computational process that reads records from the `Time_Scheduler` and produces at its output ports train ids. These are the values to a transition between the `Departure` and the `Train` instances. The rest of the behaviour of the system is then decided by the hardwired connectivity between `Train` instances on the one hand and `Platform/Junction` instances on the other. This connectivity is entered as sets of transitions in separate states. Control is set from one state to another by guards set on `Train` instances in such a way as to indicate that the procedure of going through a component has ended. That is

```
guard(train1.end_port,a_connected,new_state).
```

The action of this non-transitory guard is that it preempts control to `new_state`, whenever there is one connection at least at the arrival side. The Main manifold for this scenario is shown below.

```
Main()
{
 event step1,step2, step11, step21, step22.

 auto process platf1 is
Platform(Get_Block_F,Get_Block_P,Get_Block_S,104).
 auto process junct1 is Junction(Get_Block_F,
Get_Block_J,Get_Block_S,201).

 process train1 is
Train(TrackInfo,PlatfInfo,Get_Track,Get_ATI_R,1).
 process train2 is
Train(TrackInfo,PlatfInfo,Get_Track,Get_ATI_R,1).
 process train3 is
Train(TrackInfo,PlatfInfo,Get_Track,Get_ATI_R,3).

 begin: "main starts"->stdout;
        (activate(train1), activate(train2),
         activate(departures),
         departures.train1->train1.depart,
         departures.train2->train2.depart,
         post(step21)).
 step11: (train1.next_track->platf1.next_track,
          guard(train1.l3,a_connected,step22),

          (platf1.send->) -> train1.go,
          (platf1.send->) -> train2.go,
          (platf1.send->) -> train3.go,

          (platf1.send2->) -> train1.go2,
          (platf1.send2->) -> train2.go2,
          (platf1.send2->) -> train3.go2,

          (platf1.send3->) -> train1.go3,
          (platf1.send3->) -> train2.go3,
          (platf1.send3->) -> train3.go3,

          (&platf1->) -> train1.rl,
          (&train1->) -> platf1.tr,
          (&platf1->) -> train2.rl,
          (&train2->) -> platf1.tr).

  step21: (train1.next_track->junct1.next_track,
           train1.next2->junct1.next_track2,
           guard(train1.end_comp,a_connected,step11),

          (junct1.send->) ->  train1.go,
          (junct1.send2->) -> train1.go2,
          (junct1.send3->) -> train1.go3,

          (&junct1->) -> train1.rl,
          (&train1->) -> junct1.tr).
```

```
    step22:"step22 activated"->stdout.
  }
```

5 Conclusions

This case study is concerned with the implementation of a railway system using a control-driven coordination language. There are already two implementations of a similar system; one is based on a data-flow architecture ([7]) and the other uses another control-driven language, namely ConCoord ([3]). In the data-flow model all processes act through decisions based on a Global Data Store. In that particular application, the behaviour of control processes is based upon the event action model, which in turn is based on the values of the above data structure. There is no direct reflection of the system's architecture in this, as the controller processes communicate only with the data store.

A characteristic feature of control-driven systems is that coordination is achieved through changes of states in processes or through broadcasting of events. Particularly in Manifold the Global Data Store manipulation, is left entirely to the computational processes, which reflect changes with the raising of events. To the coordinating building blocks, data and their values mean nothing. All they cope with is the handling of event occurrences with the preemption to new states, and the connection and definition of communication in streams between such components.

Regarding the issues of precision, synchronization and interaction with the environment, the entities requiring timing for changing their state, rely on coordination processes, with a request-reply event system. For every route component that the train has to cross, there is a check on the components' status before the train is allowed to proceed. The event-triggering mechanism of Manifold provides a natural synchronization mechanism for these types of control applications. The timing constraints imposed can exhibit (soft) real-time behaviour (albeit for lack of space we have not elaborated in detail on this issue in this paper).

Finally, Manifold's philosophy on coordinators being treated as black boxes, aware only about what is happening in their immediate vicinity — namely their input and output ports — provides a natural way for achieving extensibility and adaptability (dynamic reconfiguration of the railway topology). Each component in the system (trains, junctions and platforms) operates in a quite autonomous manner, continuously consulting and updating the global state and communicating with the other components by means of events. Thus, a train only has to deal with its immediate challenge (crossing a junction or approaching a platform), while asynchronously the global state may be changing, i.e. the railway topology may be altered, the route of the train may be modified, etc.

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. F. Arbab, ‘The IWIM Model for Coordination of Concurent Activities’, *First International Conference on Coordination Models, Languages and Applications (Coordination’96)*, Cesena, Italy, 15-17 April, 1996, LNCS 1061, Springer Verlag, pp. 34-56.
2. E. de Jong, ‘Software Architectures for Large Control System: A Case Study Description’, *Second International Conference on Coordination Models, Languages and Applications (Coordination’97)*, Berlin, Germany, 1-3 Sept., 1997, LNCS 1282, Springer Verlag, 1997, pp. 150-156.
3. A. A. Holzbacher, M. Perin, M. Suhold, ‘Modelling Railway Control Systems Using Graph Grammars: A Case Study’, *Second International Conference on Coordination Models, Languages and Applications (Coordination’97)*, Berlin, Germany, 1-3 Sept., 1997, LNCS 1282, Springer Verlag, 1997, pp. 172-186.
4. 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.
5. 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.
6. 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.
7. S. Stuurman and J. van Katwijk, ‘Evaluation of Software Architectures for a Control System: A Case Study’, *Second International Conference on Coordination Models, Languages and Applications (Coordination’97)*, Berlin, Germany, 1-3 Sept., 1997, LNCS 1282, Springer Verlag, 1997, pp. 157-171.