Control of a manufacturing cell using a generic layered architecture

Man Lin
Department of Computer
and Information Science
Linköping University
S–581 83 Linköping, Sweden
linma@ida.liu.se

Jacek Malec
Department of Computer Engineering
Mälardalens Högskola
Box 883
S–721 23 Västerås, Sweden
jacek.malec@mdh.se

Abstract

This paper describes an implementation of a control system for an experimental two-robot manufacturing cell. The implementation is based on a generic layered software architecture which supports both periodic and event-driven computations. The use of the layered software architecture as the framework for designing and implementing hybrid controllers contributes to the ease of system development and is therefore strongly advocated. In order to illustrate the approach some of the implementation details are also presented.

1 Introduction

By the term *embedded systems* one usually refers to computer programs embedded in physical systems. They provide control to mechanical, chemical and other kinds of plants. The design and implementation of embedded systems usually involve the contribution from both control and computer engineers.

Simple embedded systems might be designed and implemented using rather straightforward approach. The dynamics of the system in hand needs to be studied and a control algorithm may be inferred possibly using some appropriate tool, like differential equations. An implementation of such algorithm amounts to choosing an appropriate execution model (usually periodic computations) and a computer platform, and then generating code for that platform that provably (this issue is unfortunately often omitted) implements the algorithm.

As systems become more and more complex, a plant can not be any longer described as a set of (linear) differential equations. Firstly, such a description would take too much time to create; secondly, it would be of no use (due to complexity) for the purpose of creating a control algorithm. In such cases techniques like hierarchical description and modularization come to use.

One of the modern ways to approach the problem is to use the hybrid system models: a system is described by a family of *modes*, where each mode is described as a set of differential equations [2]. The supervisory control consists of switching among modes, depending on the state of the plant. Such systems are called *hybrid* since they consist of continuous subsystems and a discrete supervisory structure. To implement hybrid controllers one needs to provide the algorithms for each control mode and the discrete controller to perform the mode switching and the interaction between the continuous and the discrete control.

Implementing hybrid controllers using languages like C or Ada is rather cumbersome since they give very poor support for algorithms expressed using state machines. Software support and computer aided tools are therefore necessary to assist the design of hybrid controllers. In this paper, we introduce a generic layered software architecture (GLA) [12] for implementing hybrid controllers. The languages and tools developed for the architecture support both periodic and discrete computations required by hybrid controllers.

This paper focuses on the implementation of a manufacturing cell control system using the layered architecture GLA.

The paper is organized as follows. First we introduce the generic layered architecture and motivate its use for designing and implementing hybrid controllers. Then we provide a brief description of how the experimental manufacturing cell looks like. Later we describe the implementation of the manufacturing cell control system with the layered architecture. The paper ends with some conclusions.

2 The generic layered software architecture

Our approach to hybrid system design and implementation builds on the three-layered software architecture developed in our lab [12] (recently renamed to GLA, the generic layered software architecture). The architecture groups similar types of computations into *layers* (shown in Figure 1).

The first layer, called the *process layer* (PL), is intended to host implementations of numerical, periodic tasks, such as identification or control. Data handled by this layer is contained in input and output vectors. Computations have the form of mappings from input vectors to output vectors and are performed

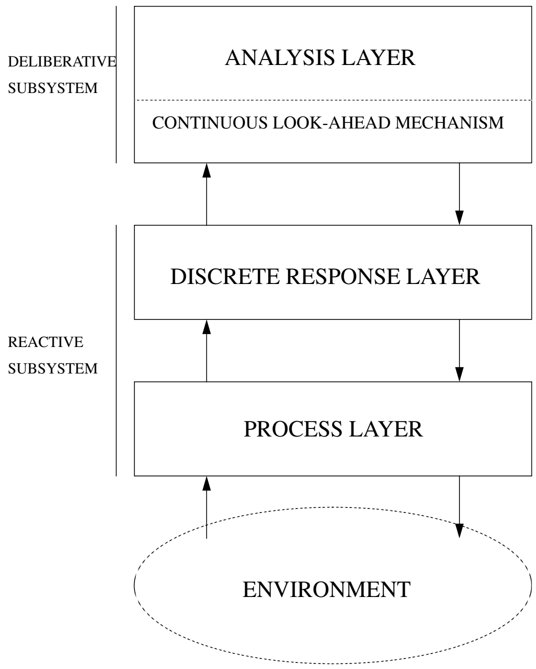

Figure 1: The generic layered software architecture.

periodically in synchronization with the sample rates of sensors.

The middle layer is called the *discrete response layer* (DRL), and performs tasks which are by nature asynchronous. For instance, it computes the response to asynchronous events that are recognized in the PL. An example of such a task is the change of control mode (of the PL) due to the change of mode in the environment. The computational model assumed for the DRL is that of discrete event systems (DES). There exist several equivalent DES formalisms: automata, transition systems, rule-based systems, etc. All of them distinguish the notions of *state* and *transition* as central, although the details vary from model to model. We have used the rule-based approach for the purpose of specifying knowledge-based system prototypes. However, this does not preclude usage of other approaches [7, 8].

The last layer is called the *analysis layer* because it is intended to handle *symbolic* reasoning tasks such as prediction, planning and scheduling, which require reference to physical time. The output of this layer can be either control events that guide the DRL in its decisions or parameter settings that are passed through the DRL and directly affect the operation of the PL.

3 Real-Time Software Tools

During our previous research we have thoroughly studied this architecture and its implications on software engineering issues [12]. One of the conclusions was that it facilitates prototyping of systems, especially because it allows development of generic software tools which can be used for implementation of each particular application system. Along this line we have developed software kernels, or *engines*, for development and implementation of the process layer and the discrete response layer. The set of tools includes:

- Process Layer eXecutive (PLX) [9]: a multi-threaded time-triggered real-time engine for implementation of process layer software. It has been implemented on a pSOS$^+$ based system and on a PC running VDX. There also exists a simulator of the PLX which runs under Unix;
- Process Layer Configuration Language
  (PLCL) [10] and its compiler: a language for specification of PL module interconnections and interfaces to both sensors and actuators on one side, and to the DRL software on the other side. The modules themselves are programmed in a subset of some conventional language, such as C or C++;
- Rule Layer eXecutive (RLX) [11]: an engine for implementation of rule-based discrete-event systems in the DRL. It has been implemented on Unix-based machines and on a PC with VDX;
- Rule Language (RL) [11]: a rule-based language for declarative specification of discrete-event control.

3.1 The Process Layer Executive

The PLX supports the implementation and maintenance of the hard real-time parts of the application which perform the transformations of periodic data.

During processing, all data is stored in a *dual state vector*, which is a global data structure consisting of an input and an output vector. The values in the input vector represent either sensor readings or internal state, whereas the values in the output vector represent actuator outputs or new internal state.

A PL application defines a sequence of transformations that should be applied to the input vector in order to compute a new output vector. Since sensor values are read periodically, the transformations have to be applied with the same periodicity. When the transformations have been applied, actuator outputs are flushed to devices and the new internal state is installed in the input vector, thus establishing the new state of the process layer.

The PLX engine supervises a PL application, which includes managing the internal state, reading sensors and writing to actuators using user defined access functions, and supervise the execution of the periodical transformations of the dual state vector. The PLX supports decomposition of the vector into several sub-vectors, each of which has its own period, causing transformations to be executed at different rates.

The PLX engine and the PLCL compiler together form the basis for a worst case execution timing analysis of a PL application [9].

3.2 The Rule Layer Executive

The rule layer executive supports the implementation of rule-based event processing. In the RLX the state of the world is represented by the time dependent values of a set of symbolic state variables called *slots*. A slot is updated only when its value changes, due to an external event or as a result of a change of another slot's value. Rules specify dependencies between slots, and typically have the following form: *if the value of a particular slot is changed in a certain way, then the value of another slot should also be changed as a result*. Internally, each such change may trigger additional rules, which lead to more updates. This forward chaining process may continue in several steps.

We take an object-oriented view of the rule base in the sense that we associate a set of rules with each slot. This view facilitates the flexibility and maintainability needed for complex systems.

The RLX tool has two major tasks. Firstly, it maintains the set of slots and rules which determines the behaviour of the discrete response layer. Secondly, it performs the forward chaining of rules. The forward chaining is typically initiated by an event recognized by the PL. The result of the forward chaining process may be the change of control algorithm used by some PL process or direct output to a device interfaced by the PL.

The RL is essentially a syntactic variant of a simple temporal logic and is used for declarative specification of behaviour of the discrete response layer. A comprehensive set of tools for correctness and consistency checking, for timing analysis of RL programs and for code generation have been developed during the recent years [4, 6, 5].

4 The manufacturing cell

In our experiments we have used a simple manufacturing cell model built using two manipulators and two conveyor belts. They are intended to model the situation illustrated in Figure 2. A similar setting has been used by Puri [13] to illustrate the hybrid system concepts with a concrete example. Such setting is also considered a benchmark for the purpose of control software verification [1, 3].

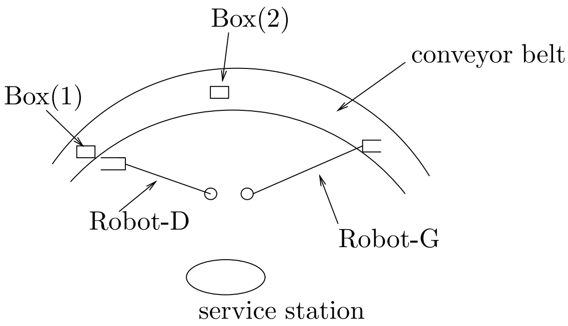

Figure 2: A two-robot manufacturing cell.

The system consists of a conveyor belt, a service station, two robots and a number of boxes. When a box is ready in the service station, Robot-D picks it up and places it on the conveyor belt, where it gets inspected. Robot-G is in charge of getting the box from the moving conveyor belt and placing it on the service station. Whenever the service station receives a processed box, it pops up an unprocessed box for Robot-D to pick up.

Although such simple system might be described using classical techniques, it is natural to think of the robots, the inspection device and the service station as a hybrid system consisting of a number of modes. Each robot might be in one of the following modes: idle, moving to grip position, gripping, transporting the box to the target position, releasing, moving to the idle position. The inspection device is either idle or inspecting. Each mode of each robot requires a separate control routine implementing a certain movement trajectory. On top of these routines a discrete controller supervising mode changes has to be implemented.

5 The experimental setting

Our experimental setting is slightly different from the one described above: the service station is replaced by another conveyor belt.

The two robot arms, Robot-D and Robot-G, are products of Eshed Robot-Tec. The software environment to control a robot arm is called CncCim. The proprietary programming language is called ACL (Advanced Control Language). The robot arm may be taught some particular position via the teach pendant. One can then use the stored positions in a program controlling movements of the arm.

The two robot arms have their separate controllers. In order to provide the supervisory control a PC equipped with the GLA software running under Linux has been used. The communication with the robots was done using RS232 channels.

The control program of each robot is run in periodic fashion. In each cycle, a control program sends sensor data to to the layered control system, sleeps for 200ms, receives a command and activates the joints according to it.

The communication between the GLA (actually, the process layer of the architecture) is done in blocking manner in order to synchronize the message exchange between the robots and the supervisor.

In this particular experiment the tasks performed by the PL might be considered trivial, as all the interesting continuous control is performed by the robot controllers. However, nothing precludes porting the robot control software to the lower layer of the GLA if such necessity occurs (e.g., in order to extend the capabilities of the controllers or to change their low-level control loops).

6 Implementation

In section 2 we have introduced the generic layered software architecture, GLA, consisting of a periodic process layer PL and an event-driven discrete response layer DRL. Each layer is programmed using its programming/specification language, PLCL plus C and RL, respectively. In this section we describe in some detail how the actual programs look like.

6.1 The PLCL program

First of all we have to define the communication channels between the GLA and the two robot controllers.

```
channel sp1 <- "/dev/ttyS1"; CINIT InitPort;
            CGET InFromPort; CPUT OutToPort;
channel sp0 <- "/dev/ttyS0"; CINIT InitPort;
            CGET InFromPort; CPUT OutToPort;
```

`InitPort, InFromPort, OutToPort` are the initialization, input, and output procedures for the two channels.

The next step consists of defining the vectors and processes residing in the process layer. For each robot arm, we define two vectors and two corresponding processes. The first process transforms and aggregates the sensor data read from the communication channel, while the second is in charge of robot movements. Each vector is assigned its update period (expressed as multiple of the basic system period, usually 10ms). Every vector contains a set of frames (dual variables). For Robot-D, the vectors are called `V_sensorD` and `V_robotD`, respectively.

```
vector V_sensorD is
 interval 1;
 select process_P_sensorD : tSwitch;
   init registerSelector[];
   get receiveFrame[];
 frame box_1 : tString;
   init registerString[]; put sendFrame[];
 frame sensed_box_1 : int; value 0;
   init registerInt[]; get receiveFrame[];
   alias b;
 frame sensed_Sigma_d : int;  value 1;
   init registerInt[]; get receiveFrame[];
   alias d;
end -- V_sensorD
```

`process_P_sensorD` is the mode selector frame of the vector `V_sensorD`. Its value decides which mode the process `P_senseD` would be in. `box_1` is a frame to be sent to the discrete response layer. Its values are `ready, not_ready, on_belt` and
`at_hand`, indicating the position and the status of the unprocessed box. `sensed_box_1` is a frame received from Robot-D. The message has the format `b =` i (for $i \in N$). `sensed_Sigma_d` is also received from Robot-D and represents its position.

```
vector V_robotD is
 interval 1;
 select process_P_robotD : tSwitch;
   init registerSelector [];
   get  receiveFrame [];
 frame  Sigma_d : tString;
   init registerString []; put sendFrame[];
 frame  T_d : int; value 0;
   init registerInt[]; put sendFrame[];
   alias T_dr;
 frame  dot_T_d : int;  value 0;
```

```
   init registerInt[]; get receiveFrame[];
 frame c : tString; init registerString[];
   put sendFrame[]; channel sp1;
end -- V1
```

process_P_robotD is the mode selector frame of vector V_robotD. Its value decides which mode the process P_RobotD is in. Sigma_d represents the position of Robot-D. Its values are Ppickup, Pputdown, Pturn and Pbegin. These values are obtained by transforming sensed_Sigma_d using the process P_senseD. They are then sent to the DRL. T_d is a clock, while dot_T_d indicates whether the clock T_d is on or off. The clock is set to on or off by the rules in the DRL. c represents the command sent to the Robot-D. The commands are idle, putdown, pickup, turn_left or turn_right (c=I, D, U, L, R).

```
process P_senseD@V_sensorD is
  mode idle is
   module idle()
      use idle[];
  end  or
  mode act is
   module com_box1(
      in *V_sensorD#sensed_box_1 :int,
      out V_sensorD#box_1 :tString)
    use com_box1[];
   module com_sigma_d(
      in *V_sensorD#sensed_Sigma_d :int,
      out V_robotD#Sigma_d :int)
    use com_sigma_d[];
  end
 active act;
end -- P_senseD
```

The process P_senseD has two modes: idle and act. The initial active mode is act. As we need to make the transformation of the input data and the value aggregation every cycle, the mode of P_senseD never changes to idle. The mode act has two modules. The first one, com_box1, computes the position of the unprocessed box (represented by box_1) while the other one, com_sigma_d, calculates the position of the Robot-D (represented by Sigma_d). The C functions that perform the transformations are called com_box1 and com_sigma_d, respectively.

The mode description for stay and pick_up for the P_RobotD process can be found in the following PLCL code. The idle mode stay is the initial mode of the system. Since the robot arm can be taught to move to a specific position, we don't need to bother with controlling the movements of the robot arm in the GLA. Instead we just send the appropriate command. Therefore, in each mode just one command is issued. Note that in the mode `d_pickup` there is an additional module `count_time` that times the execution of the gripping action.

```
process P_robotD@V_robotD is
  mode d_stay is
    module d_stay(out V_robotD#c :tString)
      use send_idle[];
  end or
  mode d_pick is
   module d_pickup(out V_robotD#c :tString)
      use send_pickup[];
   module count_pick_time(
                 in *V_robotD#dot_T_d :int,
                 out V_robotD#T_d :int)
      use count_time[];
  end or
  mode d_turnright is
   ...
  end or
  mode d_putdown is
   ...
  end or
  mode d_turnleft is
   ...
  end
active d_stay;
end -- robot_D
```

6.2 The RL program

The RL program consists of a number of *slots* and *rules*. Note that slots `process_P_robotD`, `Sigma_d`, `box_1`, `T_D` and `dot_T_d` are also defined in the process layer. They represent the mode selector, the position of Robot-D, the position of the unprocessed box, the clock and its status, respectively.

```
 --   VAR SLOT
SLOT T_d        :{0,1,2,3,4,5,6,7,8,9,10,
                  11,12,13,14,15}   := 0;
SLOT T_dr       :{0,1,2,3,4,5,6,7,8,9,10,
                  11,12,13,14,15}   := 0;
SLOT dot_T_d    :{0,1}      := 0;
SLOT Sigma_d    :{Ppickup,Pputdown,Pturn,
                  Pbegin}      := Pbegin;
SLOT box_1      :{ready,at_hand,on_belt,
```

```
                  not_ready} := not_ready;
SLOT process_P_robotD:{0,1,2,3,4,5} := 0;
```

Each rule in the rule program represents a discrete transition. A rule in the rule program consists of three parts: trigger, condition and assignment. The following listing shows some of the RL rules for the `Robot_D`.

```
-------------------------------------------
-- r1 -- robot D picks up box_1
-------------------------------------------
-- unprocessed box ready, pick up the box,
-- start to count
WHEN box_1 *= ready
IF   process_P_robotD |= 0
THEN process_P_robotD := 1, T_dr := 0,
     dot_T_d := 1;
-------------------------------------------
-- r2 -- robot D turns right
-------------------------------------------
-- after a while, Robot-D can turn right
WHEN TRUE
IF   T_d >= 14 and process_P_robotD |= 1
THEN process_P_robotD := 2, dot_T_d := 0;
```

The rule `r1` describes the transition `stay` → `pick_up` while the rule `r2` times the release of the transition `pick_up` → `turnright`. The release time is chosen to be 14 time units.

6.3 The library functions

A PLCL program refers to a number of functions that need to be provided as a separate library by the system designer. The following two simple functions illustrate the relation with the PLCL code presented earlier. Both functions are used by the mode `d_pick` of the process `P_robotD`.

```
void send_pickup(char * command)
/**----------------------------------**/
{
  strcpy(command, "U");
}       /*send c=U to the robot arm D**/

void count_time(int dt, int *t)
/**----------------------------------**/
{
  *t = *t + dt;
```

```
  fprintf(stderr,\
            "time t is %d -----", *t);
}
```

7 Conclusions

In the paper we have presented the generic layered software architecture (GLA) and its use for designing and implementing a hybrid controller for a simple manufacturing cell. We have described the basic elements of the architecture and some of the tools available (although we have omitted the more advanced tools related to the correctness and timing verification). Then we have presented fragments of the actual implementation of the controller. The simplicity of the task allowed the description to illustrate the important properties of the software without making it too lengthy or complicated. However, this doesn't mean that the capabilities of the GLA are limited to such simple cases. Recently the first author has implemented using GLA a helicopter-based traffic surveillance system which performed satisfactorily in (so far) simulation runs.

Our future work on GLA will be concentrated on guaranteeing the stability of the software engines and the associated tools. We expect to share the GLA code with the scientific community before the end of summer 1999.

8 Acknowledgments

The authors are indebted to the Applied Autonomous Sensor Systems group at the University of Örebro for allowing us to experiment with their robots and for all the help and support we have obtained during that time.

Man Lin has been supported by TFR (Swedish Research Council for Engineering Sciences) and WITAS (the Wallenberg laboratory for research on Information Technology and Autonomous Systems). Jacek Malec has been partially supported by the Mälardalens Real-Time Research Center (MRTC).

References

[1] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theorectical Computer Science*, pages 3–34, 1995.

[2] R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors. *Hybrid Systems*, LNCS 736. Springer-Verlag, 1993.

[3] C. Lewerentz and T. Lindner, editors. *Case Study "Production Cell". A Comparative Study in Formal Software Development*, FZI-Publication 1/94, Karlsruhe, 1994. Forschungszentrum Informatik.

[4] Man Lin. *Formal Analysis of Reactive Rule-based Programs*. Licentiate thesis, Linköping University, 1997. Linköping Studies in Science and Technology, Thesis No 643.

[5] Man Lin. Synthesis of control software in a layered architecture from hybrid automata. In F. W. Vaandrager and J. H. van Schuppen, editors, *Hybrid Systems: Computation and Control*, LNCS 1569, pages 152–164. Springer-Verlag, 1999.

[6] Man Lin and Jacek Malec. Timing analysis of RL programs. *Control Engineering Practice*, 6:403–408, 1998. Also in: Real-Time Programming 1997, M. Maranzana (ed.), Pergamon Press, 1997.

[7] Jacek Malec. Applied knowledge representation. *CC-AI: The Journal for the Integrated Study of Artificial Intelligence, Cognitive Science and Applied Epistemology*, 9(1):9–41, 1992.

[8] Jacek Malec. Complex behavior specification for autonomous systems. In *Proc. IEEE International Symposium on Intelligent Control'92*, pages 178–183, Glasgow, Scotland, August 1992. Also available as LiTH-IDA-R-92-14.

[9] M. Morin. Predictable cyclic computations in autonomous systems: A computational model and implementation. Licentiate thesis 352, Department of Computer and Information Sciences, Linköping University, 1993.

[10] Magnus Morin. PLCL – Process Layer Configuration Language. Technical Report LAIC-IDA-91-TR10, Linköping University, 1991.

[11] Magnus Morin. RL: An embedded rule-based system. Technical Report LAIC-IDA-94-TR2, Linköping University, 1994.

[12] Magnus Morin, Simin Nadjm-Tehrani, Per Österling, and Erik Sandewall. Real-time hierarchical control. *IEEE Software*, 9(5):51–57, September 1992.

[13] A. Puri. What are hybrid systems. http://www-path.eecs.berkeley.edu/∼anuj/-what-are-hybrid/what.html).