Towards an Integrated Framework for Multiprocessor, Multimoded Real-Time Applications *

Sergio Sáez, Jorge Real, and Alfons Crespo

Instituto de Automática e Informática Industrial,
Universitat Politècnica de València,
Camino de vera, s/n, 46022 Valencia, Spain,
{ssaez,jorge,alfons}@disca.upv.es

Abstract. In this paper we propose an approach for building real-time systems under a combination of requirements: specification and handling of operating modes and mode changes; implementation on top of a multiprocessor platform; integration of both aspects within a common framework; and connection with schedulability analysis procedures.
The proposed approach uses finite state machines to describe operating modes and transitions, and a framework of real-time utilities that implements the required behaviour in Ada 2012. Automatic code generation plays an important role: the system is derived from the functional and timing specification, and implemented according to the abstractions provided by the framework. Response time analysis enables assessing the schedulability of the different operating modes and the transitions between modes.

Keywords: Real-Time Framework, Mode Changes, Multiprocessor Scheduling, Ada 2012.

1 Introduction

The rest of this paper is organized as follows...

2 Example system

This Section describes an example system to illustrate the concepts introduced in the rest of the paper. We have chosen to use a real example system, for we believe it serves better the purpose of explaining our design choices. However, we have omitted a number of details that would only make the example harder to follow.

* This work was partially supported by the Vicerrectorado de Investigación of the Universitat Politècnica de València under grant PAID-06-10-2397

2.1 Functional description

The example system is in charge of classifying different mechanical pieces into two categories: cylinders and cubes. Figure 1 shows the example plant. Pieces are supplied to the system by means of a conveyor belt. A video camera is located on top of the conveyor belt and takes images of the first section of the belt. These images therefore reflect the input load to the system. According to that input, up to two manipulator robots will pick the pieces from the belt and place them separately depending on their type (cylinder or cube). In figure 1, this is represented by other two conveyor belts that we will not consider as a part of the system. There is also a console screen that shows information about the process.

After considering the physical elements of the system, figure 2 shows the software elements and their interconnections. A two-stage process (*segmentation* and *recognition*) analyzes the images captured by the camera. The segmentation part simply detects the type of pieces present in each image frame, and their positions on the belt. The output from the segmentation stage is inserted in the *Image_Buffer*. After segmentation, the recognition stage completes the analysis by calculating the information needed for the robots to properly catch the pieces from the belt, such as their exact orientation. The output from the recognition stage is placed in the *ToDo_Buffer*, from where they are then collected by one or two robots, controlled by tasks *Robot_0* and *Robot_1*. Each time a robot removes one piece from the belt, its corresponding robot task adds the related information to the *Done_Buffer*.

The *Graph* task extracts elements from the *Done_Buffer* and displays status information on the console screen (eg. number and type of pieces processed). Finally, the *Belt* task controls the belt speed. This task is independent from the rest of tasks, in the sense that it does not need to exchange information with them.

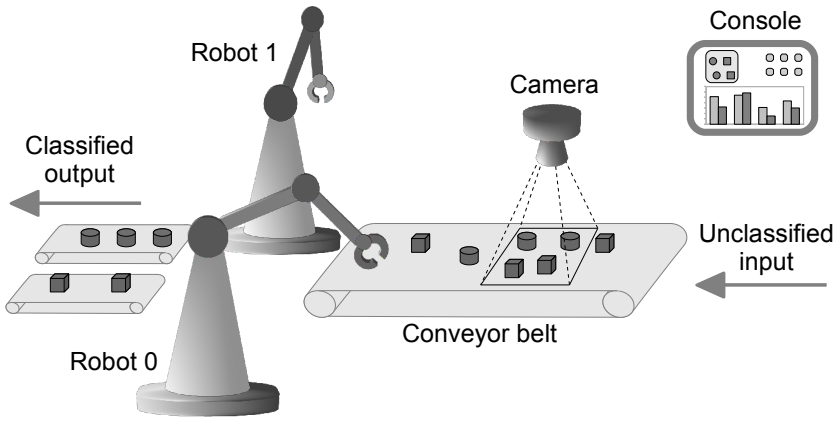

Fig. 1. View of the example plant

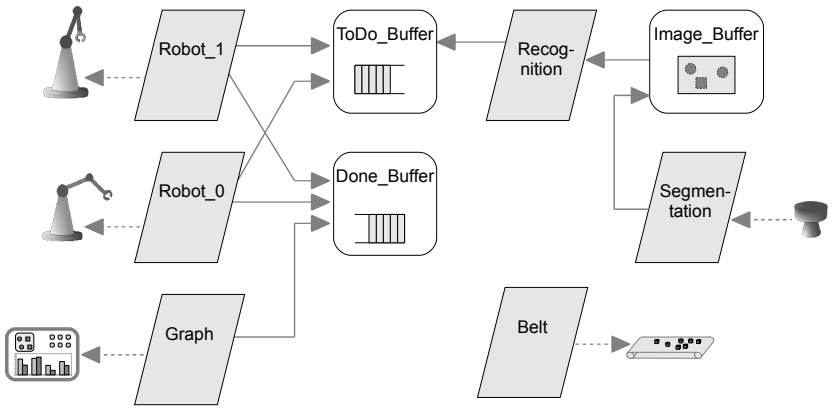

Fig. 2. Software elements of the example system

2.2 Operating modes

The flow of pieces is variable, and so is the number of pieces that remain to be removed from the belt. In order to adapt to this input variability, and to save energy and resources, three operating modes are defined for the system. The current operating mode depends on the amount of pieces that remain to be processed at a particular time interval:

Normal Mode During this mode, the number of pieces on the belt is within the range {1..Threshold}. In this situation, the platform is able to process all the pieces on the belt by using one single robot. The second robot is kept in a standby state in order to save energy. The belt advances at *normal* speed.

Overload mode When the number of pieces on the belt is greater than the threshold, the system operates in overload mode. In this mode, both robots collaborate to remove pieces from the belt, in order for the system to cope with the larger number of pieces to classify. When the number of remaining pieces is again within the threshold, the system will switch to normal mode again. By incorporating the second robot in the overload mode, we can keep the belt running at normal speed.

Fetch mode When there are no pieces to be processed on the belt (the input flow has temporarily ceased), both robots standby and the belt moves at *fast* speed in order to fetch pieces at the beginning of the belt as fast as possible. There is no need for the recognition process to run in this mode, since there are no pieces to recognize. But we still need to run the segmentation process in order to detect the arrival of new pieces. If the system is in fetch mode and then an image is taken showing a number of pieces between one and the threshold, then a switch to normal mode will occur. If the number of pieces

suddenly detected was above the threshold, then the switch is to overload mode.

2.3 Hardware platform and software workload model

Our assumed hardware platform is a two-core processor, with the two cores identified as CPU0 and CPU1. A low number of cores keeps the example simple, while it allows us to demonstrate the ability of the proposed framework to take advantage of multi-core processors.

The distribution of software tasks among processors depends on the operating mode. Table 1 shows the use of the two cores in the three defined operating modes. Note that CPU1 is in standby in modes normal and fetch, where only CPU0 is active. The tasks' names in table 1 identify the activities described in Section 2.1.

In overload mode, when the number of pieces to process is *large*, both CPUs are used for segmentation and recognition, and each CPU controls one robot. Hence we double the processing capacity for the system to cope with the input overload. More details on the timing of tasks will be given in section 3, but let us anticipate that the periods for these two tasks will be the same in normal and overload modes. We will however offset the execution of those two tasks in CPU1 by half of their period, effectively doubling the overall processing capacity.

Normal mode		**Overload mode**		**Fetch mode**	
CPU0	CPU1	CPU0	CPU1	CPU0	CPU1
`Segmentation`		`Segmentation`	`Segmentation`	`Segmentation`	
`Recognition`		`Recognition`	`Recognition`	`Belt`	
`Robot_0`		`Robot_0`	`Robot_1`	`Graph`	
`Belt`		`Graph`	`Belt`		
`Graph`					

Table 1. Distribution of tasks across CPUs and modes

Figure 3 gives the details of the workload in overload mode, which deserves further explanation. Note that both CPUs perform the same sequence of processing steps, with the difference that CPU0 executes the graph task while CPU1 controls the belt. Both CPUs execute the segmentation step at the same rate, but separated by half of the segmentation period. So they actually process different images. The segmentation task is unique, but it is scheduled to execute consecutive instances in alternate CPUs (it is *job-partitioned*).

The image buffer is split in two local shared objects, *Image_Buffer_0* and *Image_Buffer_1*. Each instance (job) of the segmentation task uses the buffer corresponding to its current CPU.

We then have two instances of the recognition task, identified in figure 3 as *Recognition_0* and *Recognition_1*. The recognition tasks suffer from input jit-

ter: they perform the recognition algorithm on the latest image taken by their corresponding segmentation task. We will ensure this by setting a deadline for segmentation equivalent to the maximum input jitter for recognition, and we will use the results of the schedulability analysis to verify that recognition always uses fresh data. We could take the same approach with recognition tasks as we have applied to segmentation, that is, to job-partition recognition among both processors, and both approaches would require an equivalent schedulability analysis. But we will make them two different tasks just to illustrate the ability of the framework to also implement such scheme. Both recognition tasks share the *ToDo_Buffer*, which is a global resource since it is used from tasks running on both processors.

Robot_0 and *Robot_1* execute in CPU0 and CPU1, respectively. They control the corresponding robots. Both tasks share the common, global resource *Done_Buffer*, where they insert information about pieces already processed. They also share the *ToDo_Buffer* among them and with the recognition tasks.

The task *Graph* collects information from the *Done_Buffer* to display statistics on the screen. In overload mode, *Belt* is an independent task running on CPU1. It is in charge of keeping the belt speed adequate to the current mode. In modes normal and fetch, *Belt* runs on CPU0. Hence this serves us to illustrate task migration between modes.

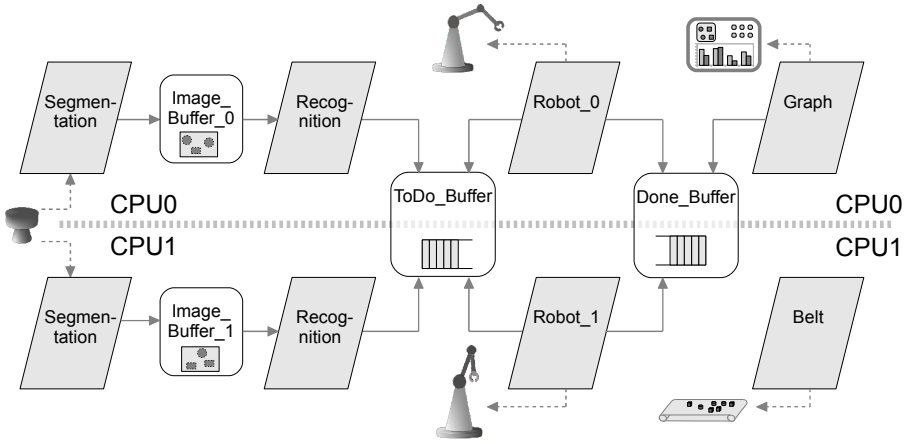

Fig. 3. Workload details in overload mode

3 Schedulability Analysis

3.1 Steady-state analysis

Table 2 shows the tasks' timing parameters and the worst-case response time analysis of the different operating modes, considered in isolation. We refer to this analysis as the *steady-state* analysis, because it does not consider transitions between modes. All tasks are periodic and the time units used are abstract. For each mode and CPU, the table shows the tasks' worst-case execution time (C), period (P), deadline (D), and input jitter (J), as well as the calculated worst-case response time (R). This response time has been obtained using the classical response time analysis equations [1, 2], with higher priorities assigned to shorter deadlines, and assuming blocking times of 2 time units for all shared resources. Note that all tasks are schedulable in the steady state, that is, in the absence of mode changes: all worst-case response times are below their respective deadlines. Table 2 also shows the utilization ratios for each CPU and mode. Note that the overload mode could not be scheduled in a single CPU, since the total utilization is greater than 100 %.

		Normal					Overload					Fetch				
CPU	Task name	C	T	D	J	R	C	T	D	J	R	C	T	D	J	R
CPU0	`Segmentation`	3	50	10		8	3	100	10		8	3	25	25		5
	`Recognition`	20	50	50	10	39										
	`Recognition_Ov`						40	100	100	10	66					
	`Robot_0`	3	10	10		5	3	10	10		5					
	`Belt`	2	50	50		10						2	25	25		2
	`Graph`	50	500	500		250	50	500	500		257	50	500	500		65
CPU0 Utilization		90 %					83 %					30 %				
CPU1	`Segmentation`						3	100	10		8					
	`Recognition_Ov`						40	100	100	10	68					
	`Robot_1`						3	10	10		5					
	`Belt`						2	50	50		10					
CPU1 Utilization		0 %					77 %					0 %				
Total utilization		90 %					160 %					30 %				

Table 2. Parameters and steady-state analysis of tasks in both CPUs and in all modes. Columns are C for worst-case execution time; T for period; D for deadline; J for input jitter; R is the calculated worst-case response time, after running the schedulability analysis.

Segmentation runs at different periods. A short period of 25 units in fetch mode, since the belt moves at *fast* speed during fetch. A medium period of 50 units in normal mode, that accommodates the *normal* speed of the belt in this mode. And a larger period of 100 units in overload. In overload mode however, the segmentation task runs in both processors. Hence the effective rate

of segmentation is the same in modes normal and overload. In mode overload, an initial offset of 50 time units for segmentation in CPU1 will ensure that segmentation runs once every 50 time units in one or the other CPU – see the mode-change analyses in Section 3.2.

Recognition requires more processing time in overload than in normal mode (40 vs. 20 units), since there are more pieces to be analyzed on the belt during an overload. The recognition task is modeled with two different task descriptions: we use `Recognition` to describe the task in mode normal, with C = 20, and `Recognition_Ov` for the overload mode, with C = 40. Note that this is just an analysis artifact: our model for mode-change analysis allows changes in all tasks' parameters, except in C. There are several reasons that justify this approach [3].

The input jitter of 10 units for recognition tasks models the required behavior that recognition always uses the freshest image pre-processed by segmentation. This input jitter is set equal to the deadline for segmentation tasks. In practical terms, the *Image_Buffer*s behave like a one-item stack, written with push and read with a blocking pop. So recognition is blocked until there is a new item in the *Image_Buffer*.

3.2 Transition analyses

Figure 4 shows the possible transitions between modes for the example system. A total of 5 transitions are possible, and they must be all analyzed for schedulability. The transition analysis must be applied to both CPUs, hence there is a total of 5 transitions × 2 CPUs = 10 analyses to consider. Some of them are however trivial. For example, all mode switches in CPU1 are schedulable because there is only one active mode in that CPU: there are no old-mode tasks when switching from normal or fetch to overload; there are no new-mode tasks in a switch from overload to normal; and there are no tasks at all involved in switches between normal and fetch in CPU1. Hence all transitions are guaranteed by the steady-state analysis on CPU1.

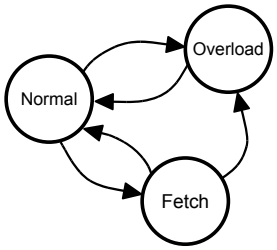

Fig. 4. Possible mode transitions

From the system description given in Table 2, we analyze the schedulability of transitions using the algorithms proposed in [3]. We have slightly modified

those tools to enable setting an initial offset in the new mode, since we needed that for our segmentation and recognition tasks in CPU1 in mode overload. The tool analyzes the transition and, if it is not schedulable, it finds appropriate offsets to new-mode tasks so that no deadlines are missed in the mode switch. For tasks with a changing period, this offset is relative to the time when the mode change request occurs. For tasks that keep their activation pace across modes, the offset is relative to the first activation of the task on the new mode.

There is no space available here for showing all the mode-change analyses results in detail. We will just note that all transitions proved schedulable after applying appropriate offsets when needed. Table 3 shows the results for a particular transition in CPU0, from overload to normal mode, before applying any offsets to new-mode tasks (the new mode is *normal*). Under these circumstances, new-mode tasks recognition and graph are not schedulable.

	Old mode tasks						
Task name	P	C	T	D	R^{MC}	X	Sched?
Robot_0	4	3	10	10	5	0	Yes
Segmentation	3	3	100	10	8	0	Yes
Recognition_Ov	2	40	100	100	79	1	Yes
Graph	1	50	500	500	490	1	Yes
	New mode tasks						
Task name	P	C	T	D	R^{MC}	Offset	Sched?
Robot_0	5	3	10	10	5		Yes
Segmentation	4	3	50	10	8	0	Yes
Belt	3	2	50	50	14	0	Yes
Recognition	2	20	50	50	114	0	**No**
Graph	1	50	500	500	740		**No**

Table 3. Analysis of transition from overload (old mode) to normal (new mode) before applying offsets. P is priority; C is worst-case execution time; T is period; D is deadline; R^{MC} is the worst-case response time during the mode-change transition; X is the worst-case phasing of the mode change request for old-mode tasks; Offset is the delay since the mode change request until the incorporation of the new-mode task. Empty offset cells mean that the corresponding task executes unchanged since the old- and new-mode periods are the same. The Sched column shows whether or not the task is schedulable during the transition.

Table 4 shows the schedulability analysis results for the transition from overload to normal after applying offsets to some new-mode tasks, in order to make the transition schedulable. There are other offset assignments that make the transition schedulable. For example, applying an offset of 64 units to recognition and a Z offset of 79 units to graph. This assignment however does not take into account the precedence relationship between segmentation and recognition. It is preferable to offset both tasks by the same amount to keep their activations

synchronized. The offset on task graph, given as Z=76, is an offset relative to the first activation of task graph in the new mode (normal).

	Old mode tasks						
Task name	P	C	T	D	R^{MC}	X	Sched?
Robot_0	4	3	10	10	5	0	Yes
Segmentation	3	3	100	10	8	0	Yes
Recognition_Ov	2	40	100	100	76	1	Yes
Graph	1	50	500	500	350	1	Yes
	New mode tasks						
Task name	P	C	T	D	R^{MC}	Offset	Sched?
Robot_0	5	3	10	10	5		Yes
Segmentation	4	3	50	10	8	50	Yes
Belt	3	2	50	50	8	0	Yes
Recognition	2	20	50	50	50	50	Yes
Graph	1	50	500	500	250	Z=76	Yes

Table 4. Analysis of transition from overload (old mode) to normal (new mode) with offsets. The transition is schedulable by offsetting both segmentation and recognition by 50 time units with respect to the mode-change request instant, and by delaying the first activation of graph 76 units with respect to its activation instant. The indication Z=76 denotes that the offset is relative to the task's first activation time in the new mode, instead of being relative to the mode-change request time.

4 High-level system specification

5 Implementation within the Real-Time Framework

6 Conclusions

References

1. Joseph, M., Pandya, P.: Finding response times in a real-time system. British Computer Society Computer Journal **29**(5) (1986) 390–395
2. Audsley, N., Burns, A., Richardson, M., Tindell, K., Wellings, A.J.: Applying new scheduling theory to static priority pre-emptive scheduling. Software Engineering Journal **8**(5) (1993) 284–292
3. Real, J., Crespo, A.: Mode Change Protocols for Real-Time Systems: A Survey and a new Proposal. Real-Time Systems **26**(2) (March 2004) 161–197