An Evaluation Framework for Complex Industrial Real-Time Embedded Systems

Yue Lu, Thomas Nolte
Mälardalen Real-Time Research Centre (MRTC), Västerås, Sweden
yue.lu@mdh.se

***Abstract*—In this technical report, we introduce an evaluation framework which are centering around four base models, inspired by an industrial robotic control application. Specifically, such evaluation models are quite complicated from a task execution and temporal dependencies perspective, making difficult to perform the corresponding timing analysis.**

I. Overview

In today's world, industrial real-time embedded systems (IRTES) are becoming more and more complex. In fact, most of such systems tend to be probabilistic in nature, which makes difficult to perform the corresponding timing analysis. This technical report introduces an evaluation framework containing 32 evaluation models, which are inspired by one example of such IRTES. Specifically, in these evaluation models, there are interesting yet intricate task execution and temporal dependencies, such as asynchronous message-passing and globally shared state variables, which may decide important control-flow conditions with major impact on task execution time as well as task response time and, 2) runtime changeability of priorities and periods of tasks and, 3) task offsets. In addition, our evaluation framework is running on a simulation framework *RTSSim*, which allows for simulating models describing the complicated control flow in tasks from both the functional and temporal behavior of systems. More details about RTSSim can be found in the following section. Finally, Section III describes the 32 evaluation models which are centering around four base models, in terms of system architecture, some interesting timing characteristics and the model source code in our simulation framework.

II. RTSSim Simulation Framework

The evaluation framework in this work is described by the modeling language in the RTSSim simulation framework [1], which allows for simulating system models containing detailed intricate execution dependencies between tasks, such as asynchronous message-passing, globally shared state variables, and runtime changeability of priority and period of tasks. In RTSSim, the system consists of a set of tasks, sharing a single processor. RTSSim provides typical RTOS services to the simulation model, such as Fixed-Priority Preemptive Scheduling (FPPS), Inter-Process Communication (IPC) via message queues, and synchronization (semaphores). The tasks in a model are described by using C functions, which are called by the RTSSim framework. The framework provides an isolated "sandbox", where time is represented in a discrete manner using an integer simulation clock, which is only advanced explicitly by the tasks in the simulation model, using a special routine, *EXECUTE*. Calls to this routine models the tasks' consumption of CPU time.

All time-related operations in RTSSim, such as timeouts and activation of time-triggered tasks, are driven by the simulation clock, which makes the simulation result independent of process scheduling and performance of the analysis PC. The response time and execution time of tasks are measured whenever the scheduler is invoked, which happens for example at IPC, task switches, *EXECUTE* statements, operations on semaphores, task activations and when tasks end. This, together with the simulation clock behavior, guarantees that the measured response time and execution time are exact.

In RTSSim, a task may not be released for execution until a certain non-negative time (the offset) has elapsed after the arrival of the activating event. Each task also has a period, a maximum arrival jitter, and a priority. Periods and priorities can be changed at any time by any task in the application, and offset and jitter can both be larger than the period. Tasks with equal priorities are served on a first come first served basis. The framework allows for three types of selections which are directly controlled by simulator input data:

- selection of execution times (for *EXECUTE*),
- selection of task-arrival jitter,
- selection of task control flow, directly or indirectly based on environmental input stimulus.

In addition, Monte Carlo simulation can be realized by providing randomly generated (conforming to the uniform distribution) simulator input data, and gives output in terms of a set of traces, each of which contains the measured Response Time (RT) and Execution Time (ET) data of each task invocation during simulation.

III. Evaluation Models

This section is split into three parts: Section III-A firstly introduces, in detail, the four base models, upon which the other evaluation models are. Next, Section III-B describes the variations of the base models which are developed by using system evolution scenarios, before the source code of the models in RTSSim is given in Section III-C.

A. Four Base Models

As we mentioned previously, the evaluation framework consisting of 32 evaluation models, are based around four different *base models*, which are designed to include some behavioral mechanisms adopted by an industrial robotic control system. Specifically, the characteristics of the behavioral mechanisms in our evaluation models include intricate task execution and temporal dependencies, e.g., asynchronous message-passing by sending and receiving messages from buffers (as shown in lines 1 to 4 in Figure 1), *execute* statements representing some computation time taken by the (sub-)task (as shown in Line 6 in Figure 1), Global Shared State Variables (GSSVs) used in selecting control branches in tasks, runtime changeability of task priorities and periods (as shown in lines 8 to 13 in Figure 1), and task offsets. Moreover, we have applied system evolution scenarios (to be introduced in the following section) to these base models to create more evaluation models.

```
1        msg = recvMessage(MyMessageQueue);
2        while (msg != NO_MESSAGE){
3              process_msg(msg);
4              msg = recvMessage(MyMessageQueue);}
5
6        execute(for_some_time);
7
8        if (GSSV1 == 1){
9          var1 = 10;
10         tcb->period = 20000;}
11       else{
12         var2 = 5;
13         tcb->period = 10000;}
```

Figure 1. Iteration-loop wrt. message passing and GSSVs, and runtime changeability of task priorities and periods in the tasks in the industrial robotic control system evaluated in our work.

It is interesting to stress that due to the existence of intricate task execution and temporal dependencies in IRTES, an upcoming RT data may not be independent of the RT data previously recorded at the same system execution, for the same task. Furthermore, the timing behavior of the adhering tasks is also quite complicated. Figure 2 shows an example, i.e., given a large[1] number at sampling, the *Probability Density Function* (PDF) histogram of the CTRL task (i.e., the most important task under analysis) RT sample in the evaluation model MV4-1 (and the relevant model source code is shown in Section III-C4) is clearly conforming to a multi-modal distribution having several peaks. Particularly, because of such distinctive feature of our target CIRTES, it is difficult to bring conventional statistical methods [3] (e.g., t-test, z-test and analysis of variance (ANOVA)) into the context of predicting the worst-case timing behavior of the

[1]By running the evaluation model MV4-1 (which is one of the most interesting and complicated evaluation models) for the time up to the upper bound on the simulation time, i.e., $2^{31}-1$, we have collected 2 000 000 sample elements of the CTRL task RT population, of which the sample size is sufficiently enough to represent the underlying population [2].

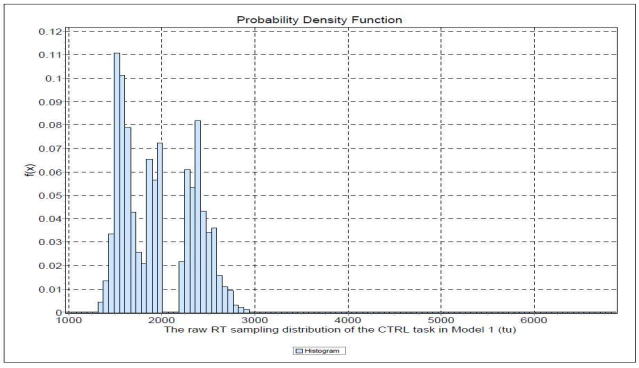

Figure 2. The Probability Density Function (PDF) histogram of a RT sample of the CTRL task in MV4-1, i.e., the most complicated evaluation model.

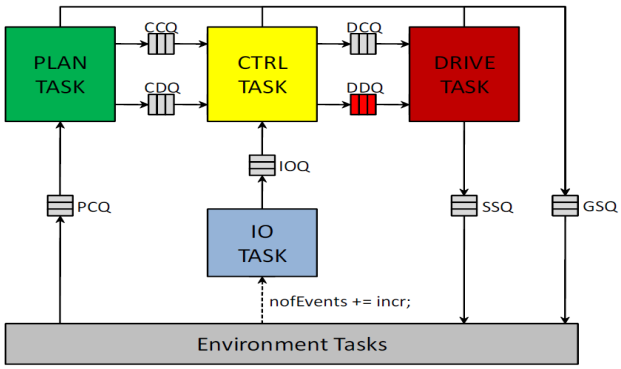

Figure 3. An industrial robotic control system architecture. From the view point of the number of queues, the main difference between MV4-* and other evaluation models is that there is no Queue SSQ in the other models.

CTRL task in IRTES as shown in [4]. Since one important assumption, i.e., the underlying population is assumed to follow a normal distribution cannot be satisfied.

Besides, the four base models are designed to have increasing complexity. To be specific, the differences between these base models are mainly concerning the contained task execution and temporal dependencies as well as the number of sub-tasks, queues and GSSVs, which are increased from MV1 to MV4, as shown in Table I, making them more complex. The system architecture of the most complicated base model MV4, is shown in Figure 3.

B. Model Variations Based upon System Evolution

In order to have a large number of evaluation models, we simply apply some typical system evolution scenarios to the four base models, each resulting in a set of new system models to analyze. In doing this, we either increase or decrease the execution time of sub-tasks in tasks in our base models, which reflects the scenarios of the change on system CPU speed. In practice, such scenarios can happen e.g., when the system is ported to a new hardware platform,

Table I
MODELS DESCRIPTION AND THE RELEVANT COMPLEXITY. THE LOWER NUMBERED COMPLEXITY IS LESS COMPLEX, I.E., 1 STANDS FOR THE SIMPLEST EVALUATION MODEL.

Models	Sub-tasks	Queues	GSSVs	Description	Complexity
MV1*	40	7	8	IPC via the bounded number of messages and GSSVs, and task offsets.	1
MV2*	42	7	8	IPC via the bounded number of messages and GSSVs, and run-time changeability of priorities of tasks, and task offsets.	2
MV3*	42	7	8	IPC via the bounded number of messages and GSSVs, and run-time changeability of priorities and periods of tasks, and task offsets.	3
MV4*	59	12	10	IPC via the *unbounded* number of messages and GSSVs, and runtime changeability of priorities and periods of tasks, and task offsets.	4

which thereby is upgraded or downgraded according to new design requirements.

- For the increase in system CPU speed, we limit ourselves to use 2, 5 and 10 as relevant factors in the work.
- For the decrease in system CPU speed, we limit ourselves to use 0.9, 0.8, 0.7 and 0.6 as relevant factors in the work. It is worth noticing that a factor of 0.5 will result in the corresponding known best-practice task WCRT longer than the corresponding task period, i.e., if one would assume the deadline of a task to be equal to its corresponding period, applying a factor of 0.5 would violate schedulability of the system.

Each of the factors highlighted above is marked as a postfix to the model name, i.e., MV1-2 represents that the CPU speed of the model variation MV1-2 is two times as fast as the CPU speed of the model MV1; In other words, the corresponding execution time of sub-tasks in MV1-2 is 0.5 times as large as the corresponding ones in MV1.

To summarize, the task parameters used in these evaluation models are shown in Table II, where the *CTRL* task is the task under analysis, which is also the task with the most complicated timing behavior.

C. Pseudo Code of Our Evaluation Models

1) $MV1-*$*:*
2) $MV2-*$*:*
3) $MV3-*$*:*
4) $MV4-*$*:*

Table II
TASKS AND TASK PARAMETERS FOR EVALUATION MODELS. THE LOWER NUMBERED PRIORITY IS MORE SIGNIFICANT, I.E., 0 STANDS FOR THE HIGHEST PRIORITY. CTRL_H AND CTRL_L REPRESENT THE CTRL TASK WITH A HIGHER AND A LOWER PRIORITY RESPECTIVELY.

Task	*Period (μs)*	*Offset (μs)*	*Priority*	*Models*
DRIVE	2 000	12 000	2	MV1-*, MV2-*, MV3-*, MV4-*
CTRL_H	20 000	0	4	MV2-*, MV3-*, MV4-*
IO	5 000	500	5	MV1-*, MV2-*, MV3-*, MV4-*
CTRL_L	10 000	0	6	MV1-*, MV2-*, MV3-*, MV4-*
PLAN	80 000	0	8	MV1-*, MV2-*, MV3-*, MV4-*

```
1  void DRIVE_TASK(TCB* tcb)
2  {
3    int msg;
4    msg = recvMessage(tcb, DDQ, 0);
5    execute(tcb, cDRIVEdecode);
6
7    switch(msg)
8    {
9      case MSG_SLC:
10       execute(tcb, cDRIVEslc);
11       if (ismoving == 0)
12       {
13         ismoving = 1;
14       }
15       break;
16
17     case MSG_SLCD:
18       execute(tcb, cDRIVEslcd);
19       if (ismoving == 1)
20       {
21         ismoving = 0;
22       }
23       break;
24     default:
25       break;
26   }
27
28   msg = recvMessage(tcb, DCQ, 0);
29   if (msg > -1)
30   {
31     switch(msg)
32     {
33       case MSG_GETSTS:
34         execute(tcb, cDRIVEgetsts);
35         sendMessage(tcb, GSQ, MSG_STS_DRIVE, FOREVER);
36         break;
37       default:
38         break;
39     }
40   }
41 }
```

Figure 4. The RTSSim code of the DRIVE task in $MV1-*$.

REFERENCES

[1] J. Kraft, "RTSSim - A Simulation Framework for Complex Embedded Systems," Mälardalen University, Technical Report, Mar. 2009.

[2] D. S. Moore, G. P. Mccabe, and B. A. Craig, *Introduction to the practice of statistics*, 6th ed. New York, NY 10010: W. H. Freeman and Company, 2009.

[3] "t-test and ANOVA, http://mathworld.wolfram.com, 2010."

```
void IO_TASK(TCB* tcb)
{
  int status;
  int eventsToProcess = 0;

  if (nofEvents > 12)
  {
    eventsToProcess = 12;
  }else{
    eventsToProcess = nofEvents;
  }

  while(eventsToProcess-- > 0)
  {
    execute(tcb, cIOEvent);
    nofEvents--;
    status = sendMessage(tcb, IOQ, 1, 0);
  }
}
```

Figure 5. The RTSSim code of the IO task in $MV1-*$.

[4] Y. Lu, J. Kraft, T. Nolte, and I. Bate, "A statistical approach to simulation model validation in response-time analysis of complex real-time embedded systems," in *The 26th ACM Symposium on Applied Computing (SAC'11)*. ACM, Mar. 2011.

```
void CTRL_TASK(TCB* tcb)
{
  int msg = -1;
  int ioevent;
  int i;
  int nofIOEvents = 0;

  msg = recvMessage(tcb, CCQ, 0);
  execute(tcb, cCTRLdecode);

  if (msg > -1)
  {
    switch (msg)
    {
      case MSG_GETSTS:
        sendMessage(tcb, DCQ, MSG_GETSTS, FOREVER);
        execute(tcb, cCTRLgetsts);
        sendMessage(tcb, GSQ, MSG_STS_CTRL, FOREVER);
      break;
      default:
        break;
    }
  }

  nofIOEvents = IOQ->current_size;

  for (i=0; i < nofIOEvents; i++)
  {
    ioevent = recvMessage(tcb, IOQ, 0);

    if (ioevent > -1)
    {
      execute(tcb, cCTRLioevent);
    }
  }

  nSLC = 5;
  tcb->period = 10000;

  msg = recvMessage(tcb, CDQ, 0);
  if (msg > -1)
  {
    switch(msg)
    {
      case MSG_FLC:
        if (idle == 1)
        {
          idle = 0;
        }
        while (nSLC-- > 0)
        {
          execute(tcb,cCTRLslc);
          sendMessage(tcb, DDQ, MSG_SLC, FOREVER);
        }
        break;
      case MSG_LAST:
        idle = 1;
        execute(tcb,cCTRLlast);
        break;
      default:
        break;
    }
  }

  if (idle == 1)
  {
    while (nSLC-- > 0)
    {
      execute(tcb, cCTRLslcd);
      sendMessage(tcb, DDQ, MSG_SLCD, FOREVER);
    }
  }
}
```

Figure 6. The RTSSim code of the CTRL task in $MV1-*$.

```
void PLAN_TASK(TCB* tcb){
    int nFLCs;
    int cmd;
    do
    {
      cmd = recvMessage(tcb, PCQ, 0);
      execute(tcb, cPLANdecode);

      if (cmd != -1)
      {
        switch(cmd)
        {
        case MSG_START:
          remainingFLC = 50;
          planstate = PLANSTATE_BEGIN;
          execute(tcb, cPLANstart);
          break;
        case MSG_STOP:
          planstate = PLANSTATE_IDLE;
          execute(tcb, cPLANstop);
          break;
        case MSG_GETSTS:
          execute(tcb, cPLANgetsts);
          sendMessage(tcb, GSQ, MSG_STS_PLAN, FOREVER);
          sendMessage(tcb, CCQ, MSG_GETSTS, FOREVER);
          break;
        default:
          break;
        }
      }
    }while (cmd != -1);

    switch (planstate)
    {
      case PLANSTATE_BEGIN:
        planstate = PLANSTATE_WORKING;
        if (remainingFLC < CDQSIZE)
        {
          nFLCs = remainingFLC;
        }else{
          nFLCs = CDQSIZE;
        }
        while (nFLCs > 0)
        {
          execute(tcb, cPLANflc);
          sendMessage(tcb,CDQ, MSG_FLC, FOREVER);
          nFLCs--;
          remainingFLC--;
        }
        break;
      case PLANSTATE_WORKING:
        if (remainingFLC < 4)
        {
          nFLCs = remainingFLC;
        }else{
          nFLCs = 4;
        }
        while (nFLCs > 0)
        {
          execute(tcb, cPLANflc);
          sendMessage(tcb, CDQ, MSG_FLC, FOREVER);
          nFLCs--;
          remainingFLC--;
        }
        break;
      case PLANSTATE_IDLE:
        break;
    }

    if (
      (remainingFLC <= 0 && planstate != PLANSTATE_IDLE)
      ||
      (remainingFLC > 0 && planstate == PLANSTATE_IDLE)
      )
    {
      execute(tcb, cPlanLast);
      planstate = PLANSTATE_IDLE;
      remainingFLC = 0;
      sendMessage(tcb, CDQ, MSG_LAST, FOREVER);
    }
}
```

Figure 7. The RTSSim code of the PLAN task in $MV1-*$.

```
void DRIVE_TASK(TCB* tcb)
{
  int msg;
  msg = recvMessage(tcb, DDQ, 0);
  execute(tcb, cDRIVEdecode);

  if (DDQ->current_size < MINDDQSIZE)
  {
    TCB* ctrl_task = findTCBbyName("CTRL_TASK");
    ctrl_task->prio = 2;
    execute(tcb, cDRIVEet200);
  }else{
    TCB* ctrl_task = findTCBbyName("CTRL_TASK");
    ctrl_task->prio = 4;
    execute(tcb, cDRIVEet100);
  }

  switch(msg)
  {
    case MSG_SLC:
      execute(tcb, cDRIVEslc);
      if (ismoving == 0)
      {
        ismoving = 1;
      }
      break;
    case MSG_SLCD:
      execute(tcb, cDRIVEslcd);
      if (ismoving == 1)
      {
        ismoving = 0;
      }
      break;
    default:
      break;
  }

  msg = recvMessage(tcb,DCQ,0);
  if (msg > -1)
  {
    switch(msg)
    {
      case MSG_GETSTS:
        execute(tcb, cDRIVEgetsts);
        sendMessage(tcb,GSQ,MSG_STS_DRIVE,FOREVER);
        break;
      default:
        break;
    }
  }
}
```

Figure 8. The RTSSim code of the DRIVE task in $MV2-*$.

```
void IO_TASK(TCB* tcb)
{
  int status;
  int eventsToProcess = 0;

  if (nofEvents > 12)
  {
    eventsToProcess = 12;
  }else{
    eventsToProcess = nofEvents;
  }

  while(eventsToProcess-- > 0)
  {
    execute(tcb, cIOEvent);
    nofEvents--;
    status = sendMessage(tcb, IOQ, 1, 0);
  }
}
```

Figure 9. The RTSSim code of the IO task in $MV2-*$.

```
void CTRL_TASK(TCB* tcb)
{
  int msg = -1;
  int ioevent;
  int i;
  int nofIOEvents = 0;

  msg = recvMessage(tcb, CCQ, 0);
  execute(tcb, cCTRLdecode};

  if (msg > -1)
  {
    switch (msg)
    {
      case MSG_GETSTS:
        sendMessage(tcb, DCQ, MSG_GETSTS, FOREVER);
        execute(tcb, cCTRLgetsts);
        sendMessage(tcb, GSQ, MSG_STS_CTRL, FOREVER);
      break;
      default:
      break;
    }
  }

  nofIOEvents = IOQ->current_size;
  for (i = 0; i < nofIOEvents; i++)
  {
    ioevent = recvMessage(tcb, IOQ, 0);

    if (ioevent > -1)
    {
      execute(tcb, cCTRLioevent);
    }
  }

  nSLC = 5;
  tcb->period = 10000;
  msg = recvMessage(tcb, CDQ, 0);
  if (msg > -1)
  {
    switch(msg)
    {
      case MSG_FLC:
        if (idle == 1)
        {
          idle = 0;
        }
        while (nSLC-- > 0)
        {
          execute(tcb,cCTRLslc);
          sendMessage(tcb, DDQ, MSG_SLC, FOREVER);
        }
        break;
      case MSG_LAST:
        idle = 1;
        execute(tcb, cCTRLlast);
        break;
      default:
        break;
    }
  }

  if (idle == 1)
  {
    while (nSLC-- > 0)
    {
      execute(tcb, cCTRLslcd);
      sendMessage(tcb, DDQ, MSG_SLCD, FOREVER);
    }
  }
}
```

Figure 10. The RTSSim code of the CTRL task in $MV2-*$.

```
void PLAN_TASK(TCB* tcb){
    int nFLCs;
    int cmd;
    do
    {
      cmd = recvMessage(tcb, PCQ, 0);
      execute(tcb, cPLANdecode);

      if (cmd != -1)
      {
        switch(cmd)
        {
        case MSG_START:
          remainingFLC = 50;
          planstate = PLANSTATE_BEGIN;
          execute(tcb, cPLANstart);
          break;
        case MSG_STOP:
          planstate = PLANSTATE_IDLE;
          execute(tcb, cPLANstop);
          break;
        case MSG_GETSTS:
          execute(tcb, cPLANgetsts);
          sendMessage(tcb, GSQ, MSG_STS_PLAN, FOREVER);
          sendMessage(tcb, CCQ, MSG_GETSTS, FOREVER);
          break;
        default:
          break;
        }
      }
    }while (cmd != -1);

    switch (planstate)
    {
      case PLANSTATE_BEGIN:
        planstate = PLANSTATE_WORKING;
        if (remainingFLC < CDQSIZE)
        {
          nFLCs = remainingFLC;
        }else{
          nFLCs = CDQSIZE;
        }
        while (nFLCs > 0)
        {
          execute(tcb, cPLANflc);
          sendMessage(tcb, CDQ, MSG_FLC, FOREVER);
          nFLCs--;
          remainingFLC--;
        }
        break;
      case PLANSTATE_WORKING:
        if (remainingFLC < 4)
        {
          nFLCs = remainingFLC;
        }else{
          nFLCs = 4;
        }
        while (nFLCs > 0)
        {
          execute(tcb, cPLANflc);
          sendMessage(tcb, CDQ, MSG_FLC, FOREVER);
          nFLCs--;
          remainingFLC--;
        }
        break;
      case PLANSTATE_IDLE:
        break;
    }

    if (
      (remainingFLC <= 0 && planstate != PLANSTATE_IDLE)
      ||
      (remainingFLC > 0 && planstate == PLANSTATE_IDLE)
      )
    {
      execute(tcb, cPlanLast);
      planstate = PLANSTATE_IDLE;
      remainingFLC = 0;
      sendMessage(tcb, CDQ, MSG_LAST, FOREVER);
    }
}
```

Figure 11. The RTSSim code of the PLAN task in $MV2-*$.

```
void DRIVE_TASK(TCB* tcb)
{
  int msg;
  msg = recvMessage(tcb, DDQ, 0);
  execute(tcb, cDRIVEdecode);

  if (DDQ->current_size < MINDDQSIZE)
  {
    TCB* ctrl_task = findTCBbyName("CTRL_TASK");
    ctrl_task->prio = 2;
    ctrl_task->period = 20000;
  }else{
    TCB* ctrl_task = findTCBbyName("CTRL_TASK");
    ctrl_task->prio = 4;
    ctrl_task->period = 10000;
  }

  switch(msg)
  {
    case MSG_SLC:
      execute(tcb, cDRIVEslc);
      if (ismoving == 0)
      {
        ismoving = 1;
      }
      break;

    case MSG_SLCD:
      execute(tcb, cDRIVEslcd);
      if (ismoving == 1)
      {
        ismoving = 0;
      }
      break;
    default:
      break;
  }

  msg = recvMessage(tcb,DCQ,0);
  if (msg > -1)
  {
    switch(msg)
    {
      case MSG_GETSTS:
        execute(tcb, cDRIVEgetsts);
        sendMessage(tcb,GSQ,MSG_STS_DRIVE,FOREVER);
        break;
      default:
        break;
    }
  }
}
```

Figure 12. The RTSSim code of the DRIVE task in $MV3-*$.

```
void IO_TASK(TCB* tcb)
{
  int status;
  int eventsToProcess = 0;

  if (nofEvents > 12)
  {
    eventsToProcess = 12;
  }else{
    eventsToProcess = nofEvents;
  }

  while(eventsToProcess-- > 0)
  {
    execute(tcb, cIOEvent);
    nofEvents--;
    status = sendMessage(tcb, IOQ, 1, 0);
  }
}
```

Figure 13. The RTSSim code of the IO task in $MV3-*$.

```
void CTRL_TASK(TCB* tcb)
{
  int msg = -1;
  int ioevent;
  int i;
  int nofIOEvents = 0;

  msg = recvMessage(tcb, CCQ, 0);

  execute(tcb, cCTRLdecode );

  if (msg > -1)
  {
    switch (msg)
    {
      case MSG_GETSTS:
        sendMessage(tcb, DCQ, MSG_GETSTS, FOREVER);
        execute(tcb,cCTRLgetsts);
        sendMessage(tcb, GSQ, MSG_STS_CTRL, FOREVER);
        break;
      default:
        break;
    }
  }

  nofIOEvents = IOQ->current_size;

  for (i=0; i < nofIOEvents; i++)
  {
    ioevent = recvMessage(tcb, IOQ, 0);

    if (ioevent > -1)
    {
      execute(tcb, cCTRLioevent);
    }
  }

  nSLC = 5;
  tcb->period = 10000;

  msg = recvMessage(tcb, CDQ, 0);
  if (msg > -1)
  {
    switch(msg)
    {
      case MSG_FLC:
        if (idle == 1)
        {
          idle = 0;
        }
        while (nSLC-- > 0)
        {
          execute(tcb,cCTRLslc);
          sendMessage(tcb, DDQ, MSG_SLC, FOREVER);
        }
        break;
      case MSG_LAST:
        idle = 1;
        execute(tcb,cCTRLlast);
        break;
      default:
        break;
    }
  }

  if (idle == 1)
  {
    while (nSLC-- > 0)
    {
      execute(tcb,cCTRLslcd);
      sendMessage(tcb, DDQ, MSG_SLCD, FOREVER);
    }
  }
}
```

Figure 14. The RTSSim code of the CTRL task in $MV3-*$.

```
void PLAN_TASK(TCB* tcb)
{
    int nFLCs;
    int cmd;
    do
    {
      cmd = recvMessage(tcb, PCQ, 0);
      execute(tcb, cPLANdecode);

      if (cmd != -1)
      {
        switch(cmd)
        {
        case MSG_START:
          remainingFLC = 50;
          planstate = PLANSTATE_BEGIN;
          execute(tcb, cPLANstart);
          break;
        case MSG_STOP:
          planstate = PLANSTATE_IDLE;
          execute(tcb, cPLANstop);
          break;
        case MSG_GETSTS:
          execute(tcb, cPLANgetsts);
          sendMessage(tcb, GSQ, MSG_STS_PLAN, FOREVER);
          sendMessage(tcb, CCQ, MSG_GETSTS, FOREVER);
          break;
        default:
          break;
        }
      }
    }while (cmd != -1);

    switch (planstate)
    {
      case PLANSTATE_BEGIN:
        planstate = PLANSTATE_WORKING;
        if (remainingFLC < CDQSIZE)
        {
          nFLCs = remainingFLC;
        }else{
          nFLCs = CDQSIZE;
        }
        while (nFLCs > 0)
        {
          execute(tcb, cPLANflc);
          sendMessage(tcb,CDQ, MSG_FLC, FOREVER);
          nFLCs--;
          remainingFLC--;
        }
        break;
      case PLANSTATE_WORKING:
        if (remainingFLC < 4)
        {
          nFLCs = remainingFLC;
        }else{
          nFLCs = 4;
        }
        while (nFLCs > 0)
        {
          execute(tcb, cPLANflc);
          sendMessage(tcb,CDQ, MSG_FLC, FOREVER);
          nFLCs--;
          remainingFLC--;
        }
        break;
      case PLANSTATE_IDLE:
        break;
    }

    if (
      (remainingFLC <= 0 && planstate != PLANSTATE_IDLE)
      ||
      (remainingFLC > 0 && planstate == PLANSTATE_IDLE)
      )
    {
      execute(tcb, cPlanLast);
      planstate = PLANSTATE_IDLE;
      remainingFLC = 0;
      sendMessage(tcb, CDQ, MSG_LAST, FOREVER);
    }
}
```

Figure 15. The RTSSim code of the PLAN task in $MV3-*$.

```
void DRIVE_TASK(TCB* tcb)
{
  int msg;
  msg = recvMessage(tcb, DDQ, 0);
  execute(tcb, cDRIVEdecode);

  if (DDQ->current_size < MINDDQSIZE)
  {
    TCB* ctrl_task = findTCBbyName("CTRL_TASK");
    ctrl_task->prio = 2;
  }else{
    TCB* ctrl_task = findTCBbyName("CTRL_TASK");
    ctrl_task->prio = 4;
  }

  switch(msg)
  {
    case MSG_SLC:
      execute(tcb, cDRIVEslc);
      if (ismoving == 0)
      {
        ismoving = 1;
        sendMessage(tcb, SSQ, MSG_MOVING, FOREVER);
      }
      break;
    case MSG_SLCD:
      execute(tcb, cDRIVEslcd);
      if (ismoving == 1)
      {
        ismoving = 0;
        sendMessage(tcb, SSQ, MSG_NOTMOVING, FOREVER);
      }
      break;
    default:
      break;
  }

  msg = recvMessage(tcb,DCQ,0);
  if (msg > -1)
  {
    switch(msg)
    {
      case MSG_GETSTS:
        execute(tcb, cDRIVEgetsts);
        sendMessage(tcb, GSQ, MSG_STS_DRIVE, FOREVER);
        break;
      default:
        break;
    }
  }
}
```

Figure 16. The RTSSim code of the DRIVE task in $MV4-*$.

```
void IO_TASK(TCB* tcb)
{
  int status;
  int eventsToProcess = 0;

  if (nofEvents > 12)
  {
    eventsToProcess = 12;
  }else{
    eventsToProcess = nofEvents;
  }

  while(eventsToProcess-- > 0)
  {
    execute(tcb, cIOEvent);
    nofEvents--;
    status = sendMessage(tcb, IOQ, 1, 0);
  }
}
```

Figure 17. The RTSSim code of the IO task in $MV4-*$.

```
void CTRL_TASK(TCB* tcb)
{
  int msg;
  int ioevent;
  int i;
  msg = recvMessage(tcb, CCQ, 0);
  execute(tcb, cCTRLdecode);

  if (msg > -1)
  {
    switch (msg)
    {
      case MSG_GETSTS:
        sendMessage(tcb, DCQ, MSG_GETSTS, FOREVER);
        execute(tcb, cCTRLgetsts);
        sendMessage(tcb, GSQ, MSG_STS_CTRL, FOREVER);
      break;
      default:
      break;
    }
  }
  i = 0;
  do{
    ioevent = recvMessage(tcb, IOQ, 0);
    if (ioevent > -1)
    { i++;
      execute(tcb, cCTRLioevent);}
  }while (ioevent > -1);

  if (closeToTarget == 1){
    nSLC = 10;
    tcb->period = 20000;
  } else {
    nSLC = 5;
    tcb->period = 10000;
  }

  msg = recvMessage(tcb, CDQ, 0);
  if (msg > -1)
  {
    switch(msg)
    {
      case MSG_FLC:
        if (idle == 1)
        {
          idle = 0;
        }
        while (nSLC-- > 0)
        {
          execute(tcb,cCTRLslc);
          sendMessage(tcb, DDQ, MSG_SLC, FOREVER);
        }
        break;
      case MSG_LAST:
        idle = 1;
        closeToTarget = 0;
        execute(tcb, cCTRLlast);
        break;
      default:
        break;
    }
  }

  if (idle == 1)
  {
    while (nSLC-- > 0)
    {
      execute(tcb,cCTRLslcd);
      sendMessage(tcb, DDQ, MSG_SLCD, FOREVER);
    }
  }
}
```

Figure 18. The RTSSim code of the CTRL task in $MV4-*$.