Optimizing the Implementation of Communication in Synchronous Reactive Models

Marco Di Natale
Scuola Superiore S. Anna
marco@sssup.it

Guoqiang Wang
Alberto Sangiovanni Vincentelli
University of California at Berkeley
geraldw@eecs.berkeley.edu
alberto@eecs.berkeley.edu

Abstract

A fundamental asset of a model-based development process is the capability of providing an automatic implementation of the model that preserves its semantics and, at the same time, makes an efficient use of the resources of the execution platform.

The implementation of communication between functional blocks in a synchronous reactive model requires buffering schemes and access procedures at the kernel level. Previous research has provided two competing proposals for the sizing of the communication buffer. We demonstrate how it is possible to leverage task timing information to obtain tighter bounds for the case of sporadic tasks or periodic tasks with unknown activation phase, and we propose an approach that applies to a more general model.

Furthermore, we provide the description of the data structures and constant-time access procedures for writer and reader tasks, and an implementation compliant with the OSEK OS standard.

1 Introduction

Model-based development of embedded real-time software aims at improving quality by fostering reuse and supporting high-level modeling and simulation tools. Synchronous Reactive (SR) models have been used in the design of hardware logic and more recently for modeling controls and control-dominated embedded applications. Synchronous reactive zero-time semantics is very popular because of the availability of tools for simulation and formal verification of the system properties.

An SR model can be defined in a declarative or imperative way. For our purposes, we consider the dataflow part of the commercial tool Simulink from Mathworks [9], consisting of a network of functional blocks, each computing a function on input signals arriving on input ports, sampled at discrete time instants, and producing the result on the output ports. In this paper, we focus on time-critical applications, modeled on the functional side as a network of SR blocks. On the architecture side, single-processor execution platforms with priority-based preemptive scheduling of tasks are the implementation target.

When implementing a high-level model into code, it is often important to preserve its semantics so to retain the validation and verification results. Preserving the semantics of the model may be a non-trivial task and, in some cases, even impossible for a given platform. The implementation of an SR multirate model can use a single task executing at the base rate of the system, or it can use a set of concurrent tasks, with typically one task for each execution rate, and possibly more. Single task implementations are easier to construct, but often characterized by poor resource utilization. Multi-task implementations allow for a much better schedulability of the resources, but because of the possible preemption, communication may have integrity or non-determinism problems and the implementation raises issues with respect to the preservation of the zero time execution behavior. These problems are illustrated in more detail after the relevant definitions are introduced in Section 2.

Given the selected platform, the definition of a model implementation consists of a set of tasks implementing the blocks and their priority assignment, and also of correct and efficient communication mechanisms. Communication of data between concurrent tasks that cannot be made atomic at the hardware level must be implemented using one of the following communication methods.

- **Lock-based**: when a task wants to access the communication data while another task holds a lock, it blocks on a semaphore [6]. When the lock is released, the task is restored in the ready state and can access the data.
- **Lock-free**: each reader accesses the communication data without blocking. At the end of the operation, it performs a check. If it realizes there was a possible concurrent operation by the writer and having read an inconsistent value, it repeats the operation. The number of retries can be upper bounded [3].
- **Wait-free**: readers and writer are protected against concurrent access by a replication of the communication buffers and by leveraging information about the time instants when they access the buffer or other information that constrains the access (such as priorities or other scheduling-related information).

1080-1812/08 $25.00 © 2008 IEEE
DOI 10.1109/RTAS.2008.23

IEEE computer society

Wait-free protocols have been mostly studied from the perspective of the programmers of concurrent real-time applications, interested in preserving the consistency of the data and providing the *freshest value at execution time*, meaning that each reader always obtains the latest data written by the writer task. These mechanisms cannot guarantee time determinism because the value read by the reader task depends upon the scheduling and the possible occurrence of preemption. In many control applications this is not a concern, given the robustness of the control algorithm with respect to time delays [16]. In other cases, the application may be sensitive to the ensuing jitter, therefore the implementation needs to satisfy more stringent requirements.

Wait-free mechanisms have also been the preferred choice for the implementation of semantics-preserving communication for their simplicity and efficiency. Any correct implementation of an SR model requires that the data item used by the reader be defined based on the activation time of the writer and reader tasks. Both tasks, however, are not guaranteed to start their execution at their release times, because of scheduling delays. Therefore, the selection of the data that is written or read must be delegated to the operating system (or to a hook procedure that is guaranteed to be executed at the task activation time).

At execution time, the writer and the readers will use the buffer positions defined at the time of their activation. Task priorities and deadlines ensure that each reader uses the value produced by the correct writer instance. Of course, there may be cases in which the writer produces multiple outputs before the reader is guaranteed to finish reading the data. In this case, the implementation consists of an array of buffers in which pointers (indexes) are assigned to writers and readers to find the right entry.

In general, the problem of defining a communication mechanism that preserves the SR semantics consists of two sub-goals. The first is to find the minimum number of required buffers. The next is to define the protocol that implements the read and write accesses to the data structure with the minimum buffer size requirement. The protocol is further divided into two parts: one executed at activation time, the other at the task execution time.

Review of Previous Work In [11], a three-slot asynchronous protocol is proposed for preserving data consistency with execution-time freshest value semantics in the communication between a single writer and a single reader running on a shared-memory multiprocessor. No assumptions are made on task priorities or periods. In general, three buffers are needed: one for the data being read, one for the data last written (current) by the writer and another when the latest written buffer has not been read yet, but the writer wants to write a new data. To achieve data integrity, a hardware supported Compare-And-Swap (CAS) instruction is used to assign atomically the reading position in the buffer array to reader tasks and to update the pointer to the last written value.

A one-to-one communication mechanism that preserves the SR semantics is presented in [5]. A two-position buffer, two buffer indexes and a reader execution flag are required. In the case of single processor systems, the code that updates the index variables is executed inside the kernel, at task activation time, therefore, there is no need for a CAS instruction, or a similar HW-level mechanism that ensures atomicity when swapping buffer pointers or comparing state variables. To guarantee that in the low to high priority communication the latency is exactly a one-unit delay, the mechanism in [5] requires that each writer task instance completes before the next is activated.

In the general case of multiple reader tasks, wait-free mechanisms and the corresponding buffer size can be defined by analyzing the relationship between the writer and its reader task instances. There are two main methods to do so. The **first method** consists in preventing concurrent accesses by computing an *upper bound for the maximum number of buffers that can be used at any given time by reader tasks*. This number depends on the *maximum number of reader instances that can be active at any time*. The **second method** provides buffer sizing and access protocols by using a general approach called *temporal concurrency control*. The size of the buffer can be computed by *upper bounding the number of times the writer can produce new values, while a given data item is considered valid by at least one reader*.

According to the **first method**, in the worst case, when no additional information is available, and no delays are defined on the communication links, the required size is equal to the maximum number N of reader task instances that can be active at any time (the number of reader tasks if task deadlines are less than or equal to periods), plus two more buffers that must be added for guaranteeing that the writer can safely update the latest data. This bound has been defined in [12], where the protocol in [11] was extended to a multi-reader asynchronous protocol for single writer and multiple readers systems. The bound can be further improved in the case of single processor systems [8] if additional information is available. In particular, if all reader tasks have priority lower than the writer, then only N+1 buffers are needed, N used by the readers and the one with the most recent update by the writer.

In [8] a semantics-preserving implementation of the single-writer to multiple-reader case is proposed and the buffer bound is further extended for the case of communication links with a unit delay. The proposed protocol is called the Dynamic Buffering Protocol (DBP). When unit delays are allowed on links, N+2 buffers are still demonstrably sufficient, where N is the number of reader tasks with lower priority. This bound is optimal for the general case of any arrival/execution pattern, that is, when no information is available on the writer and reader tasks except that each task instance terminates before its next activation

event, *but can be improved - even substantially, as shown in Section 4 - when information about task priorities and minimum inter-arrival times is available*.

The **second method** has been first introduced in [13] and [10] (under the name of NBW protocol), assuming as the validity time of the data the worst case execution time of a reader. An upper bound for the buffer size based on the concept of temporal concurrency control is also defined in [1] for the case of SR semantics preserving communication. The bound makes use of a different definition of the validity lifetime, based on the SR semantics. Details of this bounding method are presented in Section 3

A different approach is used, in [7], where an optimum number of buffers for the case of only periodic tasks with known activation phases is obtained by simulating the execution of the tasks in the least common multiple of their periods to compute the required bound. Although this approach is optimal, it can only be used when all tasks are periodic and when their activation phases are known.

A combination of the temporal concurrency control and the bounded number of readers approaches can be used to obtain an optimal sizing of the buffer. A combination of [12] and [13] was proposed in [14] for a non-SR semantics-preserving implementation. Reader tasks are partitioned into two groups: fast and slow readers. The buffer bound for the fast readers leverages the lifetime-based bound of temporal concurrency control, and the size bound for the slow ones leverages information on the maximum number of reader instances that can be active at any time. Overall, the space requirements are reduced. In [1] a similar approach is proposed for SR semantics-preserving implementations. However, the resulting buffer size is not optimal and still subject to improvement.

Contributions This paper provides the following contributions. First, we provide a *better bound* for the case of priority-scheduled tasks activated with a minimum inter-arrival time (sporadic) or periodically, but with unknown phases. The bound shows how it is possible to improve with respect to [8] by leveraging the additional information about the minimum inter-arrival times and the task priorities, and *applies to a more general (and practically useful) task model* than [7]. Furthermore, our bound also applies to a more general application model than the bounds presented up to now, namely by allowing the case of multiple active instances of tasks at the same time, and communication links with arbitrary delays. For fairness, the bound only improves on [8] when information on task periods and priorities is available (which is however typical of most implementations). Finally, motivated by the lack of implementation details in the other theoretically-driven methods, we offer an OSEK-compliant implementation of the buffer selection procedures characterized by *constant execution time*. This implementation can provide additional insights to experimentalists who may be interested in inserting these procedures in industrial applications.

2 Definitions and Assumptions

Each block in an SR (Simulink) model computes two functions: the output function and the state update function. In a synchronous reactive model, the execution of the block functions happens in zero-time, that is, the result is computed instantaneously at the time the block is activated. A system is an acyclic network of blocks. For each block b_i we use $b_i(k)$ to represent the kth occurrence of block b_i and we associate a sequence of activation times $a_i(n)$ where $a_i(1) < a_i(2) < a_i(3)\ldots$ and $a_i(n) \in \mathbb{R}^+$. Because of the zero-time assumption, the activation time captures the release, start and finish times of a block execution. The activation times of a block b_i can be constrained to be periodic (i.e., released at multiples of a given period T_i) or they can be event-triggered, possibly with a minimum inter-arrival time. Given time $t \geq 0$, we define $n_i(t)$ to be the number of times that b_i has occurred before or at t, that is: $n_i(t) = sup\{k | a_i(k) \leq t\}$, where the sup of the empty set is taken to be 0, so that if b_i has not occurred up to t, then $n_i(t) = 0$. We denote inputs of block b_i by i_i and outputs by o_i. If two blocks b_i and b_j are in an input-output relationship (b_j has as input the output of b_i and the output of b_j depends on its input), there is a link $b_i \rightarrow b_j$. The link can possibly carry a p-unit delay, as indicated by $b_i \stackrel{-p}{\rightarrow} b_j$, where $p \geq 1$. In case of a zero-delay link, if $i_j(k)$ denotes the input of the kth occurrence of b_j, then the SR semantics specifies that this input is equal to the output of the last occurrence of b_i before the kth occurrence of b_j, that is:

$$i_j(k) = o_i(m), \text{ where } m = n_i(a_j(k))$$

If b_i has not occurred yet, then $m = 0$ and the default value $b_i(0)$ is used. If the link has a p-unit delay, then:

$$i_j(k) = o_i(m), \text{ where } m = max\{0, n_i(a_j(k)) - p\}$$

The absence of zero-delay cycles (and the corresponding fixed point conditions) together with the definition of a partial order of execution for the blocks ensures that the semantics is always well-defined. The middle of Figure 1 illustrates the execution of a pair of blocks with the synchronous reactive zero-time semantics. The horizontal axis represents time. The vertical arrows capture the time instants when the blocks are activated and compute their outputs from the input values. In the figure, it is $i_j(k) = o_i(m)$. The bottom of

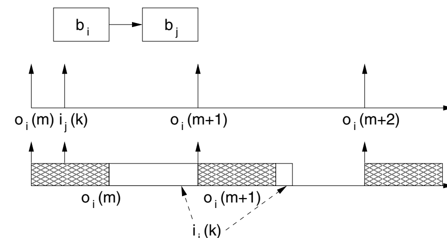

Figure 1. Preemption Changes the Values Read by the Reader

Figure 1 shows the possible problems with data transfers in multi-task implementations. A fast writer, implemented by a high priority task, communicates with a slow reader. The writer finishes its execution producing $o_i(m)$ and the reader is executed right after. If the reader performs its read operation before the preemption by the next writer instance, then $i_j(k) = o_i(m)$. Otherwise, it is preempted and a new writer instance produces $o_i(m+1)$. In case the read operation had not been performed before, the reader reads $o_i(m+1)$, in general different from $o_i(m)$. Even worse, in case the signal value is not read/written atomically, there is a finite probability that b_i preempts the reader task/block b_j while a read is in progress, resulting in a data integrity problem. The problem arises because each functional block b_i, that is, the code implementing its output and state update functions, is implemented at run time in the context of a task.

Tasks are denoted as τ_i. The block to task mapping consists of a relation between a block and a task and of a static scheduling (execution order) of the block code inside the task. $\mathcal{M}(i,k,n)$ indicates that block b_i is executed as the nth block in the context of task τ_k. For each instance of index k of task τ_i (the same notation is used for a functional block $b_i(k)$), we define as activation time $a_i(k)$ the time instant when the task is ready for execution (also defined as release time), the start time, $s_i(k)$, the time when it obtains the control of the resource, and the finish time, $f_i(k)$, when it completes its execution. The offset of $a_i(h)$ with respect to $a_j(k)$, o_{h_i,k_j}, is defined as the time separation between the release times of the two instances of the corresponding tasks, i.e. $o_{h_i,k_j} = a_i(h) - a_j(k)$. While the communication is defined among blocks, in the following, we will define bounds and rules based on the number of communicating tasks. If multiple reader blocks communicating with the same writer are mapped into the same task, they are counted as one (use the same communication buffer).

Additional task attributes are the period or minimum inter-arrival time T_i, the worst case computation time C_i, and the priority π_i. R_i is the worst case response time. Each task τ_i may also have a relative deadline d_i, smaller, equal to, or greater than T_i. When the relative deadline is larger than the period, there may be multiple instances of the same task active (but not executing) at the same time.

We assume that the runtime implementation of a block only reads the input and writes the output once for each block execution. Reads and writes can happen at any time during the execution of the block and are not atomic.

After being mapped to a task, a block is executed with the task priority. We define the following generalization of the sets in [8], where only unit delay links are considered. Each writer task can communicate with M higher priority readers. Of those, M1 are connected with unit delay links, M2 with 2-unit delays, and so on, until Mk. Also, the writer has N readers with lower priority. Of those, N0 read data with no delay, N1 with unit delay, and so on.

In any semantics preserving implementation of a link

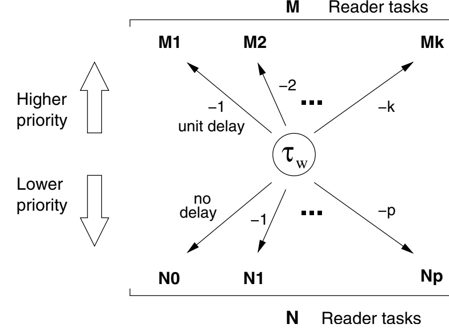

Figure 2. Writer and Reader Tasks with Link Delays and Higher/Lower Priority Levels

$b_i \to b_j$ without delay, the reader block must have a priority lower than the writer (or its scheduling must be constrained by a precedence order) [1]. In the low to high priority communication [8], it must be $d_w \leq T_w$ for the writer task, to guarantee that the latency is not higher than a one unit delay. However, when deadlines can be greater than periods, at least q units of delays are required in any link from a writer to a high priority reader

$$R_w \leq d_w \leq qT_w, \text{where } q = \left\lceil \frac{d_w}{T_w} \right\rceil.$$

3 Buffer Implementation Options

In the following, we define our improvements with respect to the sizing methods in [8] and [1] based on the number of reader instances, and on temporal concurrency control, respectively. We provide extensions to the theory and the code that implements the read and write operations in the general case of multiple-unit delays and deadlines (or better, response times) larger than periods. The proposed implementation is very efficient, as required by its kernel-level execution. For the following discussion, additional definitions are required. Each writer task is labeled as τ_{w_i}, and each reader as τ_{r_i}. NR_{w_i} and NB_{w_i} are the number of readers and the number of buffers for task τ_{w_i}, respectively.

The example shown in Table 1 consists of one writer and seven readers. It will be used to demonstrate how the proposed method improves upon the existing approaches. For simplicity, we assume that the writer has a higher priority and no delays are defined on the links. The first four columns refer to the task names, their periods, computation times and response times. The meaning of the columns from the fifth to the last will be explained in the following section. We assume the activation phases of the tasks are unknown, hence, [7] is not applicable.

3.1 Bounding the Number of Reader Instances

In [8] the writer communicates with a unit delay with M higher priority readers and N readers with lower priority. Of those, $N1$ are on a link without delay and $N2 = N - N1$

τ_i	T_i	C_i	R_i	l_i	$\left\lceil \frac{l_i}{T_w} \right\rceil$	$\left\lceil \frac{x}{T_r} \right\rceil$		$\sum_{k=1}^{i} \left\lceil \frac{x}{T_r} \right\rceil$		Eq.(3)	Eq.(6)
						R_{r_i}	l_i	R_{r_k}	l_k	$j=i$	$j=i$
τ_w	20	2	2	-	-	-	-	-	-	-	-
τ_{r_1}	8	1	3	23	2	1	3	1	3	16	8
τ_{r_2}	10	2	5	25	2	1	3	2	6	13	7
τ_{r_3}	12	2	7	27	2	1	3	3	9	10	6
τ_{r_4}	22	4	16	36	2	1	2	4	11	8	**5**
τ_{r_5}	40	4	35	55	3	1	2	5	13	**7**	**5**
τ_{r_6}	80	5	77	97	5	1	2	6	15	**7**	6
τ_{r_7}	240	10	235	255	13	1	2	7	17	13	13

Table 1. An Example Task Set

on a link with a unit delay. The data structures are illustrated in Figure 3 and the corresponding code is in Figure 4,

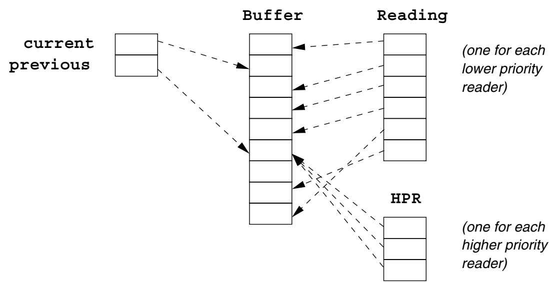

Figure 3. Data Structures for the Implementation in [8]

according to the logic outlined in [8]. The data structures consist of an array `Buffer[]` of buffers and an array `Reading[]` containing one entry for each reader task with lower priority. The entry contains the index of the buffer item used by the corresponding reader or the keyword `free`. Furthermore, two other variables contain the index of the latest written entry, `current`, and the previous one, `previous`. The code implementing the operations of the writer and reader tasks is partly executed by the kernel at task activation time and partly by the tasks at execution time. The termination code of a lower priority reader flags its termination by setting its `Reading` value as `free`.

The code to choose a free buffer (in `FindFree`) presented in [8] only states the safety criteria for free buffer selection. From the description shown in Figure 4, it seems that a quadratic time implementation is required (the *for all* statement embedded in the outer *for* loop). However, many implementations satisfying the specification are possible, with tradeoffs between space and time complexity. We provide one solution with constant execution time implementing the general case ($d \geq R$) as shown later in Figure 8. The authors of [8] demonstrate how the M high priority readers always use only one buffer (the one identified in the code by the `previous` index) and all the others require, in the worst case, a total of N+1 positions as illustrated in Figure 3. Furthermore, they assume that the deadline of each task is smaller than or equal to its period (all tasks com-

```
Data Structures
char current, previous;           char Reading[NLPREAD];
message Buffer[NBUFFERS];         char HPR[NHPREAD];
Writer
activation time                   execution time
previous = current;               ...
current = FindFree();             Buffer[current] = ...
                                  ...
FindFree() {
  for (j=0; j<NBUFFERS; j++)
    if (j != previous && for all k, Reading[k]!=j)
      return j
}
Lower Priority Reader
activation time                   execution time
if (HasDelay[i])                  ...
  Reading[i] = previous;          ... = Buffer[Reading[i]]
else                              ...
  Reading[i] = current;           Reading[i] = free;
Higher Priority Reader
activation time                   execution time
HPR[i] = previous;                ...
                                  ... = Buffer[HPR[i]]
                                  ...
```

Figure 4. Writer's and Readers' Code in [8]

plete before their deadline). Hence, only one task instance is active at any given time. In an implementation using the OSEK automotive operating system standard [4], this is indeed the case for BCC1 and ECC1 class implementations, but the case of response times larger than the period has practical relevance (as in classes BCC2 and ECC2).

A generalization to the case $R_j > T_j$ is indeed possible. When response times can be larger than the periods, the sum of the maximum number of instances of the lower priority reader tasks τ_j, that can be active at any time is given by

$$I_{w_i} = \sum_{j \in \mathbf{N}_{w_i}} I_{r_j} = \sum_{j \in \mathbf{N}_{w_i}} \left\lceil \frac{R_{r_j}}{T_{r_j}} \right\rceil,$$

where R_{r_j} is the worst case response time of the jth reader task and the set $\mathbf{N}_{w_i}$ identifies the set of reader tasks with a priority lower than writer task τ_{w_i}. In this case, considering a generalization to k-delay links, the buffer size bound can be changed to

$$NB_{w_i} = I_{w_i} + 1 + k.$$

The generalization to the case $R_j > T_j$ also requires a different set of procedures for the reader and writer tasks to access the communication data structures. The data structures need to be changed with respect to the version in [8].

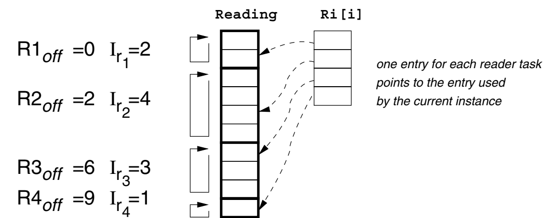

Figure 5. Data Structures for the Case $d > T$

Each reader task needs multiple entries in `Reading`, one for each instance that may be active at the same time. These may be folded into a single array, as described in Figure 5 with each reader task using a subset at (constant) offset Ri_{off} as a circular buffer of indexes (one for each instance) of size I_{r_i}. Furthermore, in case the delay on each link can be up to k units, the `current` and `previous` index variables are insufficient, and an array of $k+1$ elements is required to store the indexes referring the current and the last k elements written by the writer. The array `position` in Figure 6 may be managed as a circular buffer, with a variable `cur` marking the entry `position[cur]` of the index containing the current item and `position[(cur+j)%(k+1)]` the index of the element of `Buffer` containing the item with a j-unit delay. The array `Ri` is used to assign an index into the array `Reading` to each instance of a task at activation time. The array `Rii` is used to recover the index at task execution time (as shown in Figure 8). Finally, the array `position` provides an array of indexes to the buffer positions containing the last k+1 values written by the writer task (generalizing the case of `current` and `previous`).

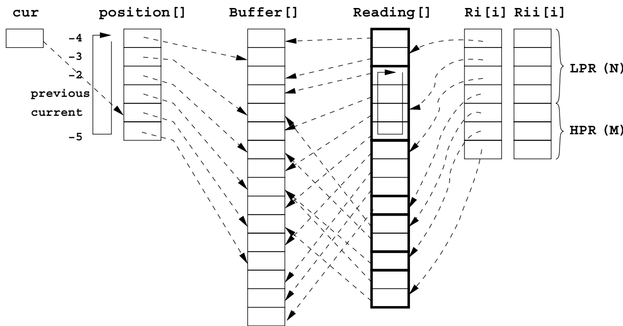

Figure 6. Data Structures for Generalized DBP

The data declaration and the new functions for the reader and writer tasks are illustrated in Figure 7 and defined in Figure 8, respectively. The array `Use` is used to keep track of how many references are currently using the corresponding entry of `Buffer`. Such references include the active reader task instances using the buffer or the possibility that the buffer contains one of the last written k+1 instances (the current and those with up to a k-unit delay). In a simple implementation, a zero value of `Use[i]` means that the entry of index i of `Buffer` is *free* and can be overwritten by the writer (we will change this implementation shortly).

The array `Use` tracks the buffers currently in use and the available ones, as shown in Figure 9. A simple linear time implementation of the routine for finding the first available buffer (`FindFreeLT`) is shown in Figure 7. However, a `for` loop executed in kernel mode at task activation time is highly undesirable and a constant time implementation of the same routine is possible. A side array can be used to

Data Structures

```
char position[k+1];
char cur, FreeB;           // position[cur] is current
message Buffer[NBUFFERS];
char Use[NBUFFERS];        // init with 0, # of users
char Reading[NINSTS];
char Ri[NR], Rii[NR];      // init with 0
```

Buffer Management Routines

```
char FindFreeLT() {
  for (j=0; j<NBUFFERS; j++)
    if (Use[j] == 0)
      return j;
}

char FindFreeCT() {
  buf_id = FreeB;
  FreeB = Use[FreeB];
  return buf_id;
}

char UseDec(char i) {
  Use[i]--;
  if (Use[i] == 0) {
    if (FreeB == -1)
      Use[i] = -1;
    else
      Use[i] = FreeB;
    FreeB = i;
  }
}
```

Figure 7. Data Structures and Routines for Generalized DBP

Writer

activation time

```
cur = (cur-1)%(k+1);              // k = max delay
UseDec(position[cur]);
position[cur] = FindFreeCT();
Use[position[cur]] = 1;
```

execution time

```
...
Buffer[position[cur]] = ···       // write
...
```

Low/High Priority Reader $j = delay[i] \in 0..k$

activation time

```
Ri[i] = (Ri[i]+1) % Ir[i];
i_id = Ri_off[i] + Ri[i];
Reading[i_id] = position[(cur+delay[i])%(k+1)];
Use[Reading[i_id]]++;
```

execution time

```
Rii[i] = (Rii[i]+1) % Ir[i];
i_id = Ri_off[i] + Rii[i];
...
··· = Buffer[Reading[i_id]];      // read
...
UseDec(Reading[i_id]);
```

Figure 8. Writer's/Readers' Code for Generalized DBP

construct a list of indexes of the free elements, as in Figure 9. The only elements of the additional array that are used to keep the list organized are the elements that match the free places in the `Use` array. Hence, the free elements of `Use` can be used to maintain the list indexes, as on the right side of Figure 9. The access routines `FindFreeCT` and `UseDec` are used to find a free buffer and release a buffer not used by any task as shown in Figure 7. `FindFreeCT` does not have to check for the availability of a buffer item, because this is guaranteed by the correct sizing of the array.

The writer has the responsibility of updating the index to the element of `position` containing the index of the buffer element that stores the next written value. Then the writer calls `UseDec` to decrement the use count indexed by `position[cur]` discarding the value with (k+1)-unit delay. After calling `FindFree` to obtain a safe buffer position `i`, the writer increments the value of `Use[i]`. Also,

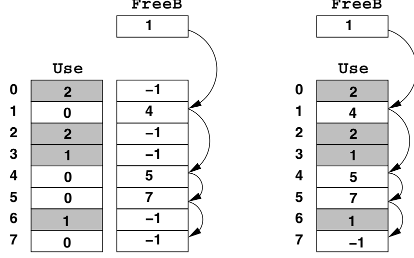

Figure 9. Constant-Time Access

readers must increment `Use[i]` whenever they are going to read data from the buffer position of index i, and decrement the `Use[i]` counter when they are done with it by calling `UseDec`. If `Use[i]` goes back to zero, the buffer is freed and added to the free list. Similar to the DBP, the termination code has to be treated as a critical section.

The code shown in Figures 7 and 8 can still be used when the buffer size is optimized as shown in the following section. The code is a general (and efficient) implementation of an SR semantics-preserving single-writer to multiple-reader communication. Before moving to the other possible methods for bounding the buffer size, we apply the DBP bound presented in [8] to our example shown in Table 1. It results in a buffer size bound of (7+1=8) items.

3.2 Temporal Concurrency Control Bound

Temporal Concurrency Control (TCC) allows to bound the number of buffers for *reader* tasks. Assume that some arrival $a_i(k)$ of a writer task instance $\tau_i(k)$ happens at time t_0 and the writer updates a buffer position of index i as shown in Figure 10. The item is used by all the instances m of the reader tasks τ_j that are activated at time $a_i(k) \leq a_j(m) < a_i(k+1)$, and it will remain valid until any reader activated in this interval has finished its execution. The following instances of the writer use buffer positions with indexes $i+1, i+2$ and so on, until, eventually, the buffer index wraps around the circular buffer and goes back to position $i-1$. The condition for a correct buffer size is that all the readers that used the previous writer buffer at position i finished using the data (the lifetime of the data value expired) when a following writer instance goes back to position i and overwrites it. This bound includes the buffer item re-

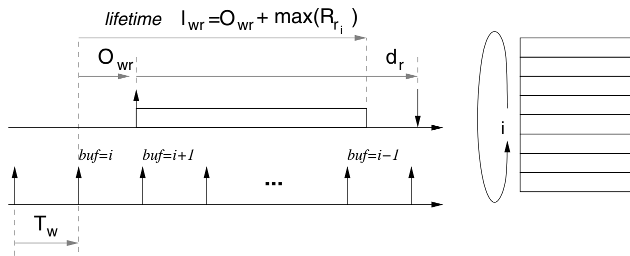

Figure 10. Buffer Sizing Based on the TCC

served to the writer for updating its current value. Define l_i as the lifetime of the data produced by the writer for reader τ_{r_i} and O_{wr} is the maximum offset between any activation of the writer and its reader τ_{r_i}. If the writer is periodic or sporadic, then $O_{wr} < T_w$. The lifetime defined in [1] is:

$$l'_{r_i} = O_{wr} + R_{r_i}.$$

If k is the delay on the communication link, then, an extension of the above definition gives the maximum lifetime for a given reader r_i:

$$l_{r_i} = kT_w + l'_{r_i} = kT_w + O_{wr} + R_{r_i}. \tag{1}$$

The response time can be computed for each reader according to the schedulability theory:

$$R_{r_i} = C_{r_i} + \sum_{j \in hp(r_i)} \left\lceil \frac{R_{r_i}}{T_j} \right\rceil C_j, \tag{2}$$

where C_{r_i} is the worst case computation time of the task containing the reader block and the summation is extended over $hp(r_i)$, which denotes all the tasks τ_j with priority higher than τ_{r_i}.

A coarse bound can be obtained by adding the buffers that are required by each reader τ_{r_j}.

$$NB_{w_i} = \sum_{\tau_{r_j} \in N_{w_i}} \left\lceil \frac{l_{r_j}}{T_w} \right\rceil$$

This estimate is too pessimistic. A better bound is obtained in [1]. For each reader, the number of required buffers is upper bound by the minimum between the number of times the writer is activated in the data lifetime and the number of reader instances that may be active in the lifetime.

$$NB_{w_i} = \sum_{\tau_{r_j} \in N_{w_i}} \min \left\{ \left\lceil \frac{l_{r_j}}{T_w} \right\rceil, \left\lceil \frac{l_{r_j}}{T_{r_i}} \right\rceil \right\}$$

This bound is further improved in [1]. The NR_{w_i} readers of τ_{w_i} are sorted by non-decreasing lifetime, so that $l_i \leq l_{i+1}$. According to temporal concurrency control, the number of buffer items:

$$NB_{w_i,j} = \left\lceil \frac{l_j}{T_w} \right\rceil$$

suffices for all the readers with lifetime $\leq l_j$. Hence, the buffer bound can be improved by dividing the reader tasks into two sets, fast and slow readers. The bound based on the data lifetime is used for the fast readers subset, and a different bound based on the number of reader activations for the remaining tasks. The reader tasks with index $i \leq j$ belong to the fast readers group. Once j is chosen, the bound on the buffer size provided in [1] is

$$NB_{w_i} = \left\lceil \frac{l_j}{T_w} \right\rceil + \sum_{i=j+1}^{NR_{w_i}} \left\lceil \frac{l_i}{T_{r_i}} \right\rceil, \tag{3}$$

where the first term represents a buffer shared among all readers such that $l_i \leq l_j$ and the second term is based on the number of reader instances inside the lifetime.

In [1] it is proposed to select the partitioning index j that defines the two groups as follows

$$j = \max\left\{ i | \left\lceil \frac{l_i}{Tw} \right\rceil \leq \sum_{k=1}^{i} \left\lceil \frac{l_k}{Tr_i} \right\rceil \right\}. \quad (4)$$

However, it is easy to show that this method leads to a suboptimal solution. To explain the consequences, in our example shown in Table 1, the partitioning index j would be the index of the last task (Equation 4), with a buffer size of 13 units as shown in the last third column of Table 1. A much better solution can be found by considering all the possible values of j in quadratic time.

$$j \in 0..NR_{w_i} | \min\left\{ \left\lceil \frac{l_j}{Tw} \right\rceil + \sum_{i=j+1}^{NR_{w_i}} \left\lceil \frac{l_i}{Tr_i} \right\rceil \right\} \quad (5)$$

In our example, there are two values of the partitioning index j, that is, $j = 5$ and $j = 6$, that minimize the right hand side of Eq.5 and results in a buffer size bound of 7 units as shown in the last second column in Table 1.

4 Improving on the Existing Bounds

Besides the possible improvement in the evaluation of the index j, expressed by Equation (5), the main problem with the bound expressed by Equations (3) and (4), however, is the definition of the number of buffers that are required by the slow reader tasks. Instead of considering the upper bound on the number of instances that can be activated for each reader task inside the lifetime and then adding them, it is better to upper bound the maximum number of (slow) reader instances that can be active at any time.

A simple example from [8] highlights the issue. The system consists of one writer and two reader tasks, with priority lower than the writer. The periods are $T_w = 2$, $T_{r_1} = 3$ and $T_{r_2} = 5$. Assuming that the response times of the readers are equal to their period, then the bound in [8] is 3 buffers, whereas the upper bound calculated according to [1] is 4.

In reality, each instance of a slow reader requires no more than one buffer at any time. For the general case, allowing for $R_i > T_i$, and delays up to k units, the bound can be defined as

$$NB_{w_i} = \left\lceil \frac{l_j}{Tw} \right\rceil + \sum_{i=j+1}^{NR_{w_i}} \min\left\{ \left\lceil \frac{R_{r_i}}{Tr_i} \right\rceil \right\} + k, \quad (6)$$

where j is chosen such that

$$j \in 0..NR_{w_i} | \min\left\{ \left\lceil \frac{l_j}{Tw} \right\rceil + \sum_{i=j+1}^{NR_{w_i}} \min\left\{ \left\lceil \frac{R_{r_i}}{Tr_i} \right\rceil \right\} + \max_{i=j+1}^{NR_{w_i}} delay[i] \right\} \quad (7)$$

This improved bound is always demonstrably better than the bounds in [8] and [1]. When the minimum value is found for $j = 0$ the bound is exactly the same as in [8]. In all the other cases, the bound is lower than the one in [1] given that it is always

$$R_i < l_i = O_{wr} + R_i,$$

and, therefore, clearly

$$\left\lceil \frac{R_{r_i}}{Tr_i} \right\rceil \leq \left\lceil \frac{l_{r_i}}{Tr_i} \right\rceil$$

For the two-task example in [8] the solution with 3 buffers is found. For our example in Table 1, the minimum number of buffer is 5 as shown in the last column in Table 1, lower than 8 (if [8] is used) and 7 (when using [1]).

5 OSEK/VDX

Starting with this section, we present an implementation of kernel-level support for SR semantics-preserving communication. To obtain portability of real-time application software, we selected OSEK/VDX as the OS platform for our implementation. The OSEK/VDX standard is widely used in the automotive industry. The implementation is an improved and revised version of the early code in [18].

To support design reuse and ease upgrade, four OSEK conformance classes are defined according to the number of active activations of a task, the task type, and the number of tasks per priority level. Based on whether they can enter a wait state by calling the `WaitEvent` service, tasks are categorized as either basic or extended. A basic task is not allowed to wait on an event. Minimum requirements for the four classes are shown in Table 2. An application utilizes kernel services by calling a set of standard APIs.

	Basic		Extended	
	BCC1	BCC2	ECC1	ECC2
Multiple Active Task Instances	No	Yes	BT: No ET: No	BT: Yes ET: No
# of Tasks not Suspended	8		16 (Any Comb. of BT/ET)	
> 1 Task/Priority	No	Yes	No	Yes
# of Events/Task	-		8	
# of Priority Levels	8		16	
Resources	RES_SCHEDULER	8(incl. RES_SCHEDULER)		
Internal Resources	2			
Alarm	1			
Application Mode	1			

Table 2. Minimum Requirements for OSEK CC

An OSEK task can be activated by either `ActivateTask` or `ChainTask` and must call `TerminateTask` for termination. An Interrupt Service Routine (ISR) has a priority level higher than that of a task. The OSEK OS standard specifies two categories of ISRs. An ISR of Category 1 is not allowed to use any kernel services and cannot be preempted. ISRs of Category 2 allow calling kernel primitives and, at the end of their execution, rescheduling occurs if there is no pending interrupt. Synchronization can be achieved by using events or semaphores. An event is owned by an extended task and

it can be set by either a task or an ISR of Category 2, but `WaitEvent` can only be invoked by extended tasks.

At least one counter is generated from a hardware or software timer. A counter is used as a time reference for alarm generation. An alarm, associated with a counter, can be used to activate a task, set an event, or call a call back routine. OSEK supports absolute and relative alarms, single instance or cyclic. Finally, the hook mechanism allows application specific functionality to be processed by the OSEK OS when kernel events occur.

OIL (OSEK Implementation Language) [17] declarations are used to configure the OS. The OIL file consists of a set of OIL objects, characterized by attributes and references and is composed of two parts: the implementation and the application definition. The former defines all standard and application-specific attributes and their properties for a particular OS implementation while the latter defines the set of objects and their corresponding attribute values for an OSEK application.

An OIL configuration file, coded manually or generated automatically, is fed to System Generator (SG), which automatically configures a kernel by choosing the required modules and customizing the data structures. The source code of the application tasks, the selected module files from the OSEK OS kernel library, and the additional application file produced by SG are compiled and linked together to produce an executable file.

6 OSEK Implementation

We describe the OSEK implementation of SR semantics-preserving protocols in a portable BCC1 conformance class. *Only standard features of OSEK are used and no modification to the kernel is required*. From Sections 3.1 and 3.2, we know that part of the protocols needs to be executed at task activation time at the kernel level. The *kernel-level* implementation is obtained by a task running with top priority or OS hook routines that cannot be interrupted and can therefore guarantee atomicity.

Task Dispatcher and Initialization In BCC1 and BCC2, events are not available and the alarm mechanism is the only way to activate periodic tasks. Since the minimum requirements allow one alarm only, we use it to periodically activate a `dispatcher` task that, in turn, activates the application tasks. The dispatcher is periodically activated by an alarm, statically configured as cyclic with a period of GCDR, the Greatest Common Divider of the application periods.

The data structures for the dispatcher are declared in Figure 11. The array `TickL[]` has dimension `LCMR`, the Least Common Multiple of the task periods. Each `TickL[i]` entry has two fields: `DispHd` and `size`. `DispHd` points to the first task on the dispatch table `DTab[]` and `size` indicates the number of tasks that need to be activated at the specific `tick` value. The array `DTab[]` contains the tasks that need to be activated from `tick = 0` to `tick = LCMR - 1`. The entries of `DTab[]` are used to index the tasks in the task descriptor array, presented in Figure 14. The right column in Figure 11 is the initialization of the data structures used by the dispatcher. At each tick `j`, the dispatch table shows the tasks that need to be activated. The fields `DispHd` and `size` are initialized to the task indexes and the number of tasks to be activated.

```
Compute TSize
char TSize = 0;
for (i=0; i<NT; i++) {
  TSize += \
    LCMR/TaskL[i].rate;
}

Declaration
struct TickEntry {
  char DispHd;
  char size;
} TickL[LCMR];

char tick;
char DTab[TSize];
```

```
Init w/o Phase Shift
tick = -1; i1 = 0;
for (j=0; j<LCMR; j++) {
  TickL[j].DispHd = -1;
  TickL[j].size = 0;
  for (i=0; i<NT; i++) {
    if (j%TaskL[i].rate==0) {
      i2 = TickL[j].size + i1;
      DTab[i2] = i;
      TickL[j].size++;
  } }
  if (TickL[j].size!=0) {
    TickL[j].DispHd = i1;
    i1 += TickL[j].size;
} }
```

Figure 11. Dispatcher Declaration and Initialization

The code in Figure 12, implemented as an initialization OSEK task or inside the main function of the application during the system startup, is responsible for the initialization of the dispatcher and communication data structures (Figure 4).

```
  ··· /* init DS and dispatcher as in Fig 11 */
  SetRelAlarm(dispAlarm, 0, GCDR);   /* set up alarm */
```

Figure 12. Initialization at System Startup

```
TASK(dispatcher) {
  tick = (tick+1) % LCMR;
  if (TickL[tick].DispHd != -1) {
    for (k=0; k<TickL[tick].size; k++) {
      idx = DTab[k+TickL[tick].DispHd];  /* task id */
      for (i=0; i<TaskL[idx].NOP; i++) { /* writers */
        idx2 = TaskL[idx].OPHd + i;
        ··· /* kernel level writer code */
    } }
    for (k=0; k<TickL[tick].size; k++) {
      idx = DTab[k+TickL[tick].DispHd];
      for (i=0; i<TaskL[idx].NIP; i++) { /* readers */
        idx2 = TaskL[idx].IPHd + i;
        ··· /* kernel level reader code */
      }
      ActivateTask(idx);
  } }
  TerminateTask();
}
```

Figure 13. Dispatcher Code

The dispatcher implementation is shown in Figure 13. During its execution, the counter `tick` is incremented modulo `LCMR`. Then, the value of the field `DispHd` of `TickL[tick]` is checked. If it is "-1", no task needs to be activated. Otherwise, the tasks in `DTab[]`, as specified by the `(DispHd, size)` are processed. For each of them, the dispatcher processes its input and output ports, perform-

ing the read or write procedures that need to be executed at the kernel level. Specifically, it calls `FindFree()` to find a safe buffer slot for each output port. For each reader, it defines the buffer slot that the task will be using during its execution. Then, the dispatcher activates the task by calling `ActivateTask` and, at the end, calls `TerminateTask` to terminate.

Communication and Task Structures Figure 14 shows the overall data structures, which are a combination of those used for the dispatcher in Figure 11 and for the communication implementation in Figure 4 with additional fields in the task descriptor. In particular, a field called `done` is added for the purpose of detecting when a context switch is executed upon the termination of a reader. A field called `owner` is added to the port descriptors. The corresponding declarations are shown in Figures 15 and 11. The

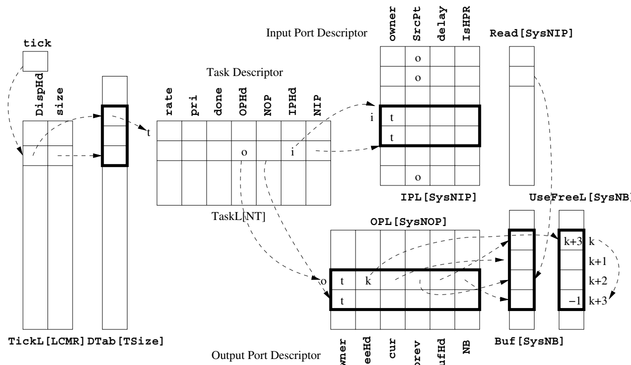

Figure 14. Data Structures

```
struct TaskEntry {      struct OPEntry {       struct IPEntry {
  char rate;              char owner;            char owner;
  char pri;               char FreeHd;           char SrcPt;
  char done;              char cur;              char delay;
  char OPHd;              char prev;             char IsHPR;
  char NOP;               char BufHd;          } IPL[SysNIP] = {
  char IPHd;              char NB;               {X,X,X,X},
  char NIP;             } OPL[SysNOP] = {        ···
} TaskL[NT] = {           {X,0,0,0,0,X},       };
  {X,X,0,X,X,X,X},        ···};
  ···};                 char UseFreeL[SysNB], Read[SysNIP];
```

Figure 15. Data Structure Declaration for DBP

task dispatcher has been shown in Figure 13: at activation time it finds and assigns a safe buffer slot to the writer and assigns the buffer index that the reader should use during execution. For the DBP, the corresponding kernel-level application code is on Figures 4 and 7. Figure 16 shows the required application-level code. We use the hook mechanism provided by OSEK to let readers atomically update the buffer free list at termination time. Specifically, we use the `PostTaskHook` to execute a critical section upon the termination of these tasks. Since the `PostTaskHook` routine executes at each context switch and for all the tasks

```
TASK (AppTask_i) {
  TaskL[i].done = 0;
  ...
  /* each writer k */
  Buf[OPL[k].cur] = ···
  ...
  /* each reader k */
  ··· = Buf[Read[k]];
  ...
  TaskL[i].done = 1;
  /* atomic hook code */
  TerminateTask();
}
```

```
void PostTaskHook(void) {
  char i,id,j,k,nip,t1,t2;
  GetTaskID(&id);
  if (id>0 && id<=NT) {
    i = id - 1;
    if (TaskL[i].done) {
      nip = TaskL[i].NIP;
      for (j=0; j<nip; j++){
        k = j+TaskL[i].IPHd;
        ···/* CS of Fig 4 */
      }
    }
} }
```

Figure 16. OSEK Implementation of an App. Task for DBP

in the system, a flag `done` is added to the task descriptor. The flag indicates for which tasks the `PostTaskHook` needs to be executed and also ensures that the operations in the `PostTaskHook` are only executed at task termination time. The `done` flag of each task is set to false at the beginning of the task, and changed to true with the last task instruction before calling `TerminateTask`.

The `PostTaskHook` routine defined in Figure 16, first obtains the identifier of the active task by calling the OSEK API `GetTaskID`. Then, it checks whether its `done` flag is set to true. If so, the updates required by the communication protocol are performed.

Implementation Complexity Table 3 shows bounds on the memory requirements for the data structures of our constant-time implementation. As for the time overhead, the implementation requires computation time for finding the free buffer and for accounting for the buffer usage, and the necessity to update the shared use list. Furthermore, since the hook mechanism is mainly designed for debugging and error management, the use of `PostTaskHook` introduces a time overhead at each context switch. The exact amount of additional time depends on the OSEK RTOS implementation and the execution platform.

char	7×NT+6×SysNOP+5×SysNIP+2×LCMR+TSize+2×SysNB+1
message	SysNB+SysNOP

Table 3. Memory Requirement Bounds

6.1 OIL Configuration File

Figure 17 shows the basic structure of an OIL configuration file. Inside the container `CPU` declaration, objects are statically specified. The application tasks are defined by the generic declaration of `AppTask_j`. We set the `SCHEDULE` attribute as `FULL`, indicating a fully preemptive scheduling policy. Under the assumption that the deadlines of application tasks are not greater than their respective periods, the `ACTIVATION` attribute is set to one (as required in BCC1). Application tasks are periodic and are activated by task `dispatcher`, therefore the attribute `AUTOSTART`

```
OIL_VERSION = "2.5";

/* Implementation Def */
IMPLEMENTATION myOSEKOS {
  ...
}; // End of myOSEKOS

/* Application Def */
CPU myCPU { // container
  /* OS Object */
  OS myOS {
    STATUS = STANDARD;
    STARTUPHOOK = FALSE;
    ERRORHOOK = FALSE;
    SHUTDOWNHOOK = FALSE;
    PRETASKHOOK = FALSE;
    POSTTASKHOOK = TRUE;
    USEGETSERVICEID = FALSE;
    USERESSCHEDULER = FALSE;
  };
  /* Task Object */
  TASK AppTask_j {
    PRIORITY = X_j;
    SCHEDULE = FULL;
    ACTIVATION = 1;
    AUTOSTART = FALSE;
  };
  ...
  TASK dispatcher {
    PRIORITY = X_d;
    SCHEDULE = NON;
    ACTIVATION = 1;
    AUTOSTART = FALSE;
  };
  /* Alarm Object */
  ALARM dispAlarm {
    COUNTER = SysTimer;
    ACTION = ACTIVATETASK{
      TASK = dispatcher;
    };
    AUTOSTART = TRUE {
      ALARMTIME = 0;
      CYCLETIME = GCDR;
      APPMODE = AppMode0;
  };};
  /* Counter Object */
  COUNTER SysTimer {
    MINCYCLE = x;
    MAXALLOWEDVALUE = x;
    TICKSPERBASE = x;
  };
  /* Appl Mode Object */
  APPMODE AppMode0 {
    VALUE = AUTO;
  };
}; // End of myCPU
```

Figure 17. OIL Configuration File

is set to FALSE. Task dispatcher activates the application tasks and performs part of the communication protocol operations on behalf of the kernel. Therefore, its priority is higher than those of all application tasks, and its SCHEDULE attribute is set to NON, indicating a non-preemptive scheduling. dispatcher is activated by an alarm, so its AUTOSTART attribute is set to FALSE. The alarm object dispAlarm is specified accordingly. The alarm is associated with a counter, which is an object defined in the OIL file. The alarm is configured to activate task dispatcher by setting its attribute ACTION as ACTIVATETASK. Finally the alarm's AUTOSTART attribute is set to TRUE and the period of dispAlarm is set to GCDR. The atomicity of the termination code that updates the shared use list is guaranteed by the PostTaskHook mechanism, which is turned on by setting the attribute POSTTASKHOOK to TRUE in the OS object as in Figure 17.

7 Conclusions

The paper presents a new bound for sizing the buffer in an SR semantics-preserving implementation of a communication channel between a writer and multiple reader tasks executing at different rates. The bound subsumes and in general improves on existing bounds. Furthermore, the current formulation is capable of handling a general model, allowing for multiple active task instances at the same time and communication links with arbitrary delays. The proposed bound is not demonstrated as optimum and further improvements may be still possible. Finally we presented an OSEK BCC1 implementation of the protocol with constant time complexity.

References

[1] M. Baleani, A. Ferrari, L. Mangeruca, and A. S. Vincentelli, "Efficient embedded software design with synchronous models," 5th ACM EMSOFT conference, 2005.

[2] M. G. Harbour, M. Klein, and J. Lehoczky, "Timing analysis for fixed-priority scheduling of hard real-time systems," IEEE Trans. on Software Engineering, vol. 20, no. 1, Jan. 1994.

[3] J. W. S. Liu, Real-Time Systems. Prentice Hall, 2000.

[4] OSEK, "Osek os version 2.2.3 specification,"

[5] N. Scaife and P. Caspi, "Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems," 6th ECRTS Conference, July 2004.

[6] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," IEEE Transactions on Computers, vol. 39, no. 9, Sept. 1990.

[7] S. Tripakis, C. Sofronis, N. Scaife, and P. Caspi, "Semantics-preserving and memory-efficient implementation of inter-task communication on static-priority or edf schedulers," 5th ACM EMSOFT Conference, Sept. 2005.

[8] C. Sofronis, S. Tripakis, and P. Caspi, "A Memory-Optimal Buffering Protocol for Preservation of Synchronous Semantics under Preemptive Scheduling," 6th ACM EMSOFT Conference, October 2006.

[9] MathWorks, "The MathWorks Real-Time Workshop User's Guide", Version 6, September 2006.

[10] H. Kopetz, J. Reisinger, "The Non-Blocking Write Protocol NBW: A Solution to a Real-Time Synchronization Problem", Proceedings of the 14th IEEE RTSS, 1993.

[11] J. Chen, A. Burns, "A Three-Slot Asynchronous Reader/Writer Mechanism for Multiprocessor Real-Time Systems", Technical Report YCS-286, University of York, January 1997

[12] J. Chen, A. Burns, "A Fully Asynchronous Reader/Write Mechanism for Multiprocessor Real-Time Systems", Technical Report YCS-288, University of York, May 1997

[13] J. Chen, A. Burns, "Loop-Free Asynchronous Data Sharing in Multiprocessor Real-Time Systems Based on Timing Properties", RTCSA Conference, pp 236-246, 1999

[14] H. Huang, P. Pillai, K. Shin, "Improving Wait-Free Algorithms for Interprocess Communication in Embedded Real-Time Systems", USENIX 2002 Conference, pp 303-316.

[15] L. Mangeruca, M. Baleani, A. Ferrari, A. L. Sangiovanni-Vincentelli, "Uniprocessor Scheduling Under Precedence Constraints", 12th RTAS Conference, 2006

[16] P. Caspi, A. Benveniste, "Toward an Approximation Theory for Computerised Control", EMSOFT 2002: 294-304

[17] OSEK Implementation Language (OIL), Version 2.5: available at http://www.osek-vdx.org/.

[18] G. Wang, M. Di Natale, A. Sangiovanni Vincentelli, "An OSEK/VDX Implementation of Synchronous Reactive Semantics Preserving Communication Protocols", 2007 OSPERT Workshop, June 2007, Pisa, Italy.