Automatic Code Generation for Synchronous Reactive Communication*

Guoqiang Wang‡
University of California
Berkeley, CA, USA
geraldw@eecs.berkeley.edu

Marco Di Natale
Scuola Superiore Sant'Anna
Pisa, Italy
marco@sssup.it

Pieter J. Mosterman
The MathWorks
Natick, MA, USA
pieter.mosterman@mathworks.com

Alberto Sangiovanni-Vincentelli
University of California
Berkeley, CA, USA
alberto@eecs.berkeley.edu

Abstract

Synchronous Reactive models are used in Model-Based Design to define embedded control applications. The advantage of Model-Based Design is that system properties can be verified on the model and applied to its software implementation if the translation of the model into code preserves its semantics. In this paper, we present an automatic code generation framework for the semantics-preserving implementation of communication in multi-rate systems. The proposed solution applies to the widely used MATLAB® and Simulink® products. It leverages the Target Language Compiler template language of Real-Time Workshop® and extends the applicability of available commercial code generators. The overhead in memory of the presented solution is analyzed and compared with other implementations.

1 Introduction

Model-Based Design (MBD) for embedded real-time software aims at fostering reuse and improving quality by capturing the functionality of the design with a model of computation and deriving implementation code from this model automatically. Further, MBD enables design-time simulation and verification of properties on models. Among the MBD formalisms to capture design functionality, Synchronous Reactive (SR) models [1] are effectively used in the design of hardware logic and for modeling control-dominated embedded applications. SR models are very popular because they permit the use of tools for fast simulation and formal verification of the system properties. They are typically characterized by the "perfect synchrony hypothesis", which requires that the system completes the reaction to an event before the occurrence of any other event.

When implementing a high-level SR model into code, it is important to preserve its semantics, so to retain the simulation and verification results. At run time, a model of communicating SR functional blocks may be implemented by a concurrent program, in which tasks have a finite execution time and are possibly subject to preemption. Semantics preservation requires that the run-time behavior is provably equivalent to the "zero execution time" behavior of the model. This property subsumes the property that model communication flows are preserved in the program implementation (*stream preservation*) [2]. In general, demonstrating that the implementation respects the semantics of the SR specification is non-trivial.

This work addresses SR models that exhibit a subset of the full Simulink® functionality. Here, a system is a network of blocks communicating through ports. A block can be a reader, a writer, or both. All blocks in the model react in response to events that may be periodic, defining a sampling rate, but may also be aperiodic. Each block computes two functions: the output and the state update functions. According to the synchronous assumption, the computation is considered to occur in zero time, meaning that the outputs and the state depend instantaneously on the input values. This implies that the input values must be determined at the time the block is activated.

For each block $\mathtt{b_i}$, let $\mathtt{b_i(j)}$ and $\mathtt{a_i(j)}$ be its $\mathtt{j^{th}}$ instance and the corresponding activation time, respectively. Under the SR semantics, given that the execution time is zero, $\mathtt{a_i(j)}$ captures also the start and the finish times of $\mathtt{b_i(j)}$. Assume writer $\mathtt{w}$ communicates with reader $\mathtt{r_i}$. Let

*This work was supported by the MARCO/DARPA Gigascale Systems Research Center (GSRC). Their support is gratefully acknowledged.

‡Now with National Instruments Corporation.

†MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks.

$o_w(j)$ and $i_{r_i}(j)$ be the output and the input of the j^{th} instance of w and r_i, respectively. We define $\zeta_i(t)$ to be the number of times that b_i has been activated up to time t, i.e., $\zeta_i(t) = \sup\{m | a_i(m) \leq t\}$, where the sup of an empty set is defined to be zero. Let $delay[i]$ be the communication link delay for reader i. The SR communication semantics can be formulated as $i_{r_i}(j) = o_w(k)$, where $k = \max\{0, \zeta_w(a_i(j)) - delay[i]\}$.

Upon implementation, the functionality of SR blocks is realized by run-time tasks. There are two options to implement an SR multi-rate model on single-processor platforms. In a single task implementation, all the SR blocks are implemented by one task running at the system base rate. Such an implementation is easier to construct, but is often characterized by poor resource utilization.

A multi-task implementation uses one task for each execution rate, and possibly more, executed under the control of an Operating System (OS). Each task is characterized by a set of parameters: priority, period (T), computation or execution time, worst case response time (R), and relative deadline (d). A task can be preempted according to its priority. Schedulability of tasks requires that $R \leq d$. Multi-task implementations allow for better schedulability. As discussed in [3], because of interference or preemption, this may possibly lead to problems with respect to communication of data streams over links, i.e., non-deterministic communication and/or data integrity problem. Figure 1 shows the difference between the model behavior (top of the figure) and a possible multi-task implementation. The k^{th} instance of b_j should use $i_j(k) = o_i(m)$ as input. However, if blocks are executed by tasks as in the bottom part of the figure, the execution of $b_j(k)$ may be delayed by interferences from higher priority tasks (hashed box), resulting in $i_j(k) = o_i(m+1)$. Even worse, if data communication is not atomic, preemption while reading/writing may cause the data integrity to be compromised. To solve these problems, different mechanisms have been proposed.

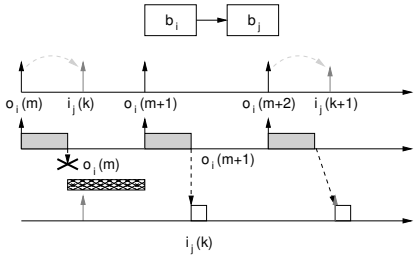

Figure 1. Issues with Semantics Preservation

Previous Work Wait-free schemes can be used to protect a writer and its readers against concurrent access to the communication data by replicating the communication buffers and possibly by leveraging knowledge of access times and scheduling constraints such as task priorities and

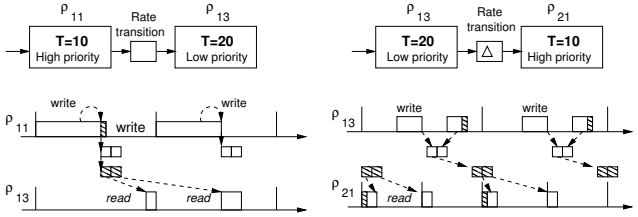

Figure 2. Rate Transition Block of Simulink®

periods. An example of a wait-free access control mechanism is the Rate Transition (RT) block of Simulink. The RT block behaves like a Zero-Order Hold block for fast to slow transitions or a Unit Delay block plus a Hold block (or Sample and Hold) for slow to fast transitions. In the fast to slow rate (high to low priority) transitions, the RT block output and update functions execute at the rate of the slower block, but with the priority of the faster block (dashed box in the left side of Figure 2). In low to high rate (priority) transitions, the RT block output update function runs in the context of the fast task, but at the rate of the slow task as the first function (dashed box in the bottom-right side of Figure 2). The RT block state update function executes in the context of the slow task as the last function and updates the internal buffer state (the striped box in the right side of Figure 2). The output function uses the state of the RT block that was updated in the previous instance of the slow task. In the case of activation with identical phase of communicating blocks with harmonic rates, the RT block guarantees data consistency and stream preservation. *However, when activations are not periodic or the writer and its readers do not have harmonic periods or are activated with different phases, RT blocks cannot be used and a more general communication mechanism is needed.*

As discussed in [3], a general buffer-based communication scheme consists of two parts: a sizing mechanism and an indexing procedure. Buffer sizing mechanisms may be based on writes that are spatially out of order or in order. The out-of-order version computes an upper bound for the maximum number of buffers that can be used at any given time by a writer and its reader task instances. In [4], an asynchronous protocol that guarantees data consistency with the freshest-value semantics between a writer and multiple readers is presented. The protocol needs $N + 2$ buffer slots, where N is the number of readers. Because of their simplicity and efficiency, wait-free schemes are also the preferred choice to implement SR communication protocols. In [5, 6], the Double Buffer (DoB) mechanism for one-to-one communication with SR semantics is presented, using a two-place buffer and two buffer indices. In the case of single processor systems, given that the code that updates the index variables is executed inside the kernel at task activation time, there is no need for a hardware mechanism to ensure atomicity when swapping buffer pointers or com-

paring state variables. In [7], the Dynamic Buffering (DyB) protocol is defined for single-writer multiple-reader systems with unit communication delay links, under the assumption that each task instance terminates before its next activation event. It is demonstrated that $\mathtt{N}+2$ buffers are sufficient for DyB, where N is the number of lower priority readers. The Constant Time Dynamic Buffering (CTDyB) protocol presented in [3] provides an efficient kernel-level implementation. The in-order version uses the notion of temporal concurrency control to compute a buffer size by upper bounding the number of times the writer can produce new data while a given data is considered valid by at least one reader. This concept is first introduced in [8, 9] for communication with the freshest-value semantics, assuming as the data validity time the worst case execution time of a reader. These two mechanisms are also used in [10] for buffer sizing while preserving the SR communication semantics.

In [11], the communication scenario presented in [7] is further generalized to handle arbitrary multi-unit communication delays and multiple instances of a task active at the same time. A bound on the number of required buffers is provided for the communication of blocks with a general pattern of activation. An efficient constant time implementation is presented as a set of C functions in [3]. The work presented here improves on the code implementation of the proposed method and defines its automation and integration with modeling in the context of the commercial Simulink and Real-Time Workshop® [12] products for modeling and automatic code generation. The result is a library of custom Simulink blocks and the template files that control the generation of the code. *The blockset extends the capability of available commercial tools by allowing stream preserving communication of aperiodic blocks and/or blocks activated with different phases and non-harmonic periods.*

An example of an application system that requires support for this type of communication is sensory fusion (e.g., as needed by active safety functions in the automotive domain). In this case, the subsystem performing the integration of sensory information processes data streams coming from heterogeneous sensors (e.g., ultrasound and radar). The rates at which sensory data arrive depend on the sensor technology and the sensor manufacturer. It is quite common to have sensors operating at non-harmonic rates (such as a 20 ms cycle for the ultrasound and 50 ms for the radar). Furthermore, the sensory fusion subsystem benefits from the time determinism on the input streams because it must correlate in time the information about objects coming from different sensor sources.

2 Protocol Implementation: DoB & CTDyB

In this section, we present two SR communication protocols (DoB and CTDyB) and we show their implementation according to the OSEK OS standard. Figure 3 shows the

Data Structure		
`boolean wrtIdx, rdIdx;`		`message buf[2];`
	Low to High Priority	**High to Low Priority**
Initialization	`wrtIdx = 0;`	`rdIdx = wrtIdx = 0;`
	`buf[0] = buf[1] = ···`	
Writer		
Activation	`wrtIdx = !wrtIdx;`	`if (rdIdx == wrtIdx)` `  wrtIdx = !wrtIdx;`
Execution	`buf[wrtIdx] = ···`	
Reader		
Activation	`rdIdx = !wrtIdx;`	`rdIdx = wrtIdx;`
Execution	`··· = buf[rdIdx];`	

Figure 3. Double Buffer Mechanism

pseudo-code of the DoB mechanism in [5, 6] for single-writer single-reader communication. In DoB, a pair of buffers and two indices are maintained. DoB takes advantages of information on the relative priority of the reader and the writer tasks. Buffers are initialized and the writing index initially points to the first buffer. The reading index of a lower priority reader is initialized in the same way. At activation time, the writing index of a lower priority writer is always toggled while the writing index of a higher priority writer is toggled only if the reading and the writing indices are equal. The reading indices of a higher and a lower priority reader are assigned as the negated and the same value of the writing index, respectively.

A pseudo-code description of the CTDyB mechanism in [3] is shown in Figure 4. As a generalization of DoB, CTDyB handles communication between a writer and multiple readers. An array of data buffers and indices to its entries are maintained. Communication links with unit delay require an additional index variable `prev`. The buffer size

Data Structure		
`char delay[NR], isHPR[NR];`		`char read[NR], UFL[NB];`
`char cur, prev, freeHd;`		`message buf[NB];`

Initialization	`cur = 0;`
	`prev = 0;`
	`freeHd = 1;`
	`buf[0] = ···`
		`UFL[0] = 2;`
		`for(j=2; j<NB-1; j++)`
		`  UFL[j] = j + 1;`
		`UFL[NB-1] = -1;`

	Writer	**Reader i**

Activation	`UseDec(prev);`
	`prev = cur;`
	`cur = FindFree();`
	`UFL[cur] = 1;`
		`if (delay[i])`
		`  read[i] = prev;`
		`else`
		`  read[i] = cur;`
		`if (isHPR[i] == 0)`
		`  UFL[read[i]]++;`

Execution	`buf[cur] = ···`	`··· = buf[read[i]];`
Termination		`if (isHPR[i] == 0)` `  UseDec(read[i]);`

Auxiliary Functions	`char FindFree() {`
	`  char t = freeHd;`
	`  freeHd = UFL[t];`
	`  return t;`
	`} /* O(1) */`
		`void UseDec(char i) {`
		`  UFL[i]--;`
		`  if (UFL[i] == 0) {`
		`  UFL[i] = freeHd;`
		`  freeHd = i;`
		`} }`

Figure 4. CT Dynamic Buffering Mechanism

is equal to the sum of the number of lower priority readers plus two. As in [3], the constant-time algorithm for finding a free entry in the array uses a list to track the free buffers and counters to maintain information on the number of readers using each entry. The first free slot on the list is always indicated by `freeHd`. At activation time, for a writer, the use count of the slot pointed by the old `prev` index is decremented and the buffer slot is freed if the use count drops to zero. Then, `prev` and `cur` are updated and the use count of the slot pointed by `cur` is set to one. Similarly, for a reader, its reading index is assigned according to its link delay and the use count of the buffer item is incremented accordingly. At termination time of a lower priority reader, the use count of the buffer item used by the reader is decremented and then freed if the counter becomes zero. For both DoB and CTDyB, at execution time, writer and reader blocks write into and read from the slots that have been assigned at their activation times, respectively. The RT block mechanism used in the code generated from Simulink by Real-Time Workshop is a special case of DoB.

OSEK Implementation Preservation of communication flows amounts to ensuring that the reader accesses the data produced by the correct instance of the writer task. In particular, the buffer slots containing the item produced by the writer and read by a reader have to be defined at their respective activation times. Both writer and reader tasks, however, are not guaranteed to start execution at their release times because of scheduling delays. Therefore, in general, the selection of the data buffer entry must be delegated to the OS (or a hook procedure that is guaranteed to be executed at task activation time). We choose the OSEK automotive standard [13] as the target OS. OSEK defines four Conformance Classes (BCC1/2 and ECC1/2) with corresponding minimum requirements. In OSEK, a task can be activated by `ActivateTask` and must call `TerminateTask` on termination. At least one counter is generated by a hardware or software timer. The counter can be 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 callback routine. OSEK supports absolute and relative alarms, single instance or cyclic.

OIL (OSEK Implementation Language) declarations are used to configure an OSEK application. In the implementation of the CTDyB protocol presented in [3], lower priority readers need to call PostTaskHook to release their buffers upon termination. However, PostTaskHook is executed at each context switch (not only at task termination), therefore adding unnecessary time overhead. In this paper, we use a different approach to handle the atomic termination of lower priority readers. Besides the CTDyB protocol, code generation for the DoB protocol is also supported for comparison with implementations using Rate Transition blocks.

Task Dispatcher Although the procedures for accessing the communication buffers can be used in the general case of block activated by arbitrary events with a minimum inter-arrival time [11], in our library blockset the block that implements the connection with the task scheduling (the task dispatcher) assumes periodic activation events. Our implementation also assumes $\mathtt{R} \leq \mathtt{T}$, which implies that only one active instance for each task exists at any time. To obtain software portability, one alarm is used in accordance with the BCC1 compliancy rules. The proposed task dispatcher is shown in Figure 5. The dispatcher is sampled at the base rate of the system. For each sampling rate (task), there is a corresponding counter. The counters are initialized appropriately so that all tasks are scheduled at startup. The counters are incremented at the system base rate and reset to 0 when they reach their periods (i.e., the corresponding task needs to be scheduled). The main functionality of the dispatcher is to assign writing and reading indices at task activation time. Depending on the chosen protocol, the dispatcher assigns buffer indices via the code in Figure 3 or 4. Then the tasks are activated by calling the OSEK API function `ActivateTask`. The dispatcher terminates by calling `TerminateTask`.

The first `if` statement in Figure 5 is for CTDyB only. At system startup, the termination segment in the dispatcher is not needed, which is achieved by checking the flag, `systemStartup`. At system startup, the flag is initialized as true and later on during execution it is reset.

Priority information is needed when assigning indices for both DoB and CTDyB. We assume that the priority assignment is based on Rate Monotonic when no information is provided by the users.

The dispatcher is functionally equivalent to the implementation in [3], but more efficient in terms of speed and memory usage. The use of `PostTaskHook` is avoided by letting the dispatcher perform the termination for lower

Task(dispatcher) {
```
  RMS();
  /* CTDyB only: LPR termination */
  if (!systemStartup) {
    for (i=0; i<NT; i++) {
      if (cntr[i] == 0) {
        ··· /* LPR termination */
    } }
    systemStartup = 0;
  }
  /* kernel-level index assignment */
  for (i=0; i<NT; i++) {
    if (cntr[i] == 0) {
      ···/* process task i's writers */
      ···/* process task i's readers */
      ActivateTask(i);
  } }
  TerminateTask();
}
```

Data Structure
```
char p[NT], cntr[NT];
```
Initialization
```
systemStartup = 1;
for (i=0; i<NT; i++) {
  cntr[i] = p[i] - 1;
}
```
Auxiliary Function
```
RMS(void) {
  for(i=0; i<NT; i++){
    cntr[i]++;
    if(cntr[i]==p[i]){
      cntr[i] = 0;
    }
  }
}
```

Figure 5. Implementation of Task Dispatcher

```
TASK(init) {
  ... /* initialization required by protocol as in Figure 3/4 */
  ... /* initialization required by dispatcher as in Figure 5 */
  SetRelAlarm(dispAlarm, actionOffset, baseRate);
  TerminateTask();
}
```

Figure 6. General Structure of Task Init

priority readers. Furthermore, the implementation of application tasks is simplified by avoiding the data structures indicating the task status and the dispatch table used in [3].

Initialization Task The data structures of the communication protocols in Figures 3 and 4 and those used by the dispatcher in Figure 5 must be initialized. These are performed by the OSEK task `init` shown in Figure 6. The task `init` also sets the cyclic alarm, `dispAlarm`, which periodically activates the task dispatcher at the system base rate. Through `SetRelAlarm` and `ActivateTask`, the scheduling of the dispatcher and application tasks is integrated with the RTOS scheduler. Task `init` only needs to execute at system startup and is terminated by `TerminateTask`.

The next step is to automatically generate the implementation by leveraging the customizable extensions of the Real-Time Workshop code generation environment.

3 Real-Time Workshop®

Real-Time Workshop (RTW) is a product for automatic code generation, packaging, and compilation from Simulink models. Two important RTW features are code generation for user-created blocks via System Function (S-Function) blocks and code customization flexibility via the Target Language Compiler (TLC) [14].

The capability of the Simulink environment can be extended by custom-defined S-Functions, coded in C or MATLAB®. Similar to built-in Simulink blocks, S-Function blocks must provide an implementation according to a pre-defined API to allow execution by the Simulink simulation engine. S-Functions are compiled by the mex utility and the compilation results are stored in Mex-files. S-Function blocks can be put in a custom-defined library.

TLC is an RTW component that enables customization of the generated code. The RTW build process converts a graphical Simulink model into an intermediate form of the block diagram, which includes all the model-specific information required for code generation. Then, TLC transforms the intermediate description into target-specific code.

TLC includes block-level and system-level files. Block-level files define the generation of code for block-level features and system-level files capture model-wide information that is used for generating header and parameter information in the program code. TLC files control the way RTW generates code and can be customized for special needs. They are indeed often used to generate efficient code for S-Function blocks. By default, S-Functions are translated into non-inlined code by the RTW code generator, which results in memory and time overhead because of a significant data structure that is created for each instance of the S-Function block in the model and the function call for invoking the block implementation. This overhead can be reduced by providing a TLC file that allows inlining the S-Function. The following section describes how to support code generation for SR communication protocols by extending RTW with custom TLC files.

4 SR Protocol Code Generation

Before delving into code generation, we first need to address data transfer at the application task level. For the protocols shown in Figures 3 and 4, indexing procedures at the kernel and application levels are presented with the corresponding data structures. As discussed in Section 2, kernel level functionality can be implemented as a task dispatcher. Similar to the Rate Transition block in Simulink, the application level indexing can be implemented as a communication buffer block that reads in and writes out data at the rates of the writer and reader, respectively.

In the design of communication data structures, we encapsulate within the S-Function buffer blocks the data that is only accessed by its code. Data that is shared between the dispatcher and the communication buffer blocks is defined as data stores (shared variables). For example, for the DoB in Figure 3, `wrtIdx` and `rdIdx` are accessed by both the dispatcher and communication buffer (on behalf of writer and reader) and thus they are represented as data stores. Similarly, for the CTDyB in Figure 4, `cur` and `read[]` are global variables. However, `prev`, `freeHd`, and `UFL[]` are local to the dispatcher and hidden inside the block implementation. For both mechanisms, `buf[]` is maintained locally by the communication buffer.

The dispatcher and the communication buffer S-Functions are coded in C and the corresponding blocks are made available as part of a custom library.

4.1 Task Dispatcher

Figure 7 shows our custom SR implementation library. The second and fourth block from the left in the top row are the task dispatchers for DoB and CTDyB, respectively. They have no input port and a variant number of output ports for writing and reading indices. The number of output ports can be customized for each block by changing its parameters from the Simulink designer GUI. The number of output ports clearly depends on the number of readers for the communication buffers. In our library, under the default configuration, the DoB dispatcher assigns indices for

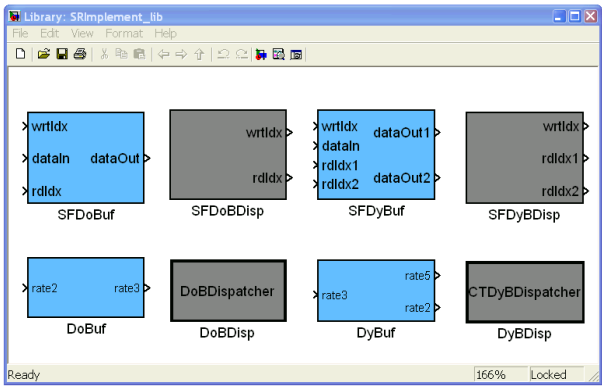

Figure 7. SR Implementation Library Blocks

a writer and its reader while the DyB dispatcher assigns indices for a writer and its two readers.

The sampling period of the task dispatcher is the system base period. Given knowledge on the priority of the writer and each reader and on the delays with each communication link, the dispatcher assigns writing and reading buffer indices to writers and readers at their respective rates.

The dispatcher block is constructed as an "atomic" block, which means that the corresponding code can be generated as a function (not inlined). This is needed because the rate grouping policy used by RTW or Real-Time Workshop® Embedded Coder™ [15] (the specialized code generator for embedded systems) would otherwise merge the dispatcher code inside a possible application task executing at the base rate.

4.2 Communication Buffer

The first and third block from the left in the top row of Figure 7 are the communication buffers for DoB and CTDyB, respectively. As discussed earlier, the communication buffer block handles shared data between a writer and its reader(s) at the application task level. In our library, the communication buffer block for DoB has three input ports (writing index, the data input, and reading index) and one output port. However, the counterpart for CTDyB has a variant number of input and output ports, depending on the number of readers it may have. Similar to DoB, each reader contributes to one input port (for reading index) and one output port (for communication data). In the custom library, the communication buffer for CTDyB under default configuration transfers data between a writer and its two readers.

Because the sampling rates of the writer and readers may be different, the sampling of the communication buffer block needs to be port-based. During execution, when the writer produces its outputs, the buffer block reads both the data value and the buffer index value assigned to the writer and stores the data into the corresponding buffer slot. Similarly, when a reader executes, the buffer block reads in the assigned reading index value and then outputs the data identified by the reading index to the output port.

4.3 Dispatcher and Shared Buffer Interaction

The last issue we need to address is how to support write and read indices shared between the dispatcher and the communication buffer S-Functions. This is accomplished by using Simulink Data Store Memory blocks. The dispatcher writes the assigned write and read indices into the data store memory via the data store write block while the communication buffers obtain the assigned indices through the data store read block on behalf of the writer and readers. With the integration of the dispatcher, the communication buffers, and the data store read/write blocks, the SR communication semantics can always be guaranteed, regardless of the relative activation rates of the writer and its readers [6, 7].

To ease model construction, the communication buffer can be masked with its corresponding data store reads into a wrapper block. Similarly, a wrapper block can be generated by grouping the dispatcher block and its corresponding data store writes. The dispatcher and communication buffer wrapper blocks are shown in the second row of Figure 7.

4.4 Model-Wide TLC File Customization

The SR communication protocols are implemented using the above discussed S-Functions and the corresponding TLC files enable generation of efficient code. However, model-wide TLC files still need to be customized to control the definition of the scheduling sampling rates, the generation of the base rate functions, and the scheduling of the tasks executing at different rates. Furthermore, an OSEK-specific TLC library file is created to generate OSEK tasks, the main function, and the OIL configuration file.

5 Code Generation and Execution Results

In this section, we use a pair of simple Simulink models to show the generated code for the communication links and the results of the code execution.

The example shown in Figure 8 is a Simulink model in which a block pair communicates using the DoB mechanism. The writer consists of an adder that adds its result at the previous sampling step and a constant of 2. The reader simply corresponds to an output port. The sampling periods of the writer and the reader are 2 and 3 seconds (not harmonic), respectively. Hence, a multi-tasking implementation using the Rate Transition block is not possible and indeed not allowed by the tool.

Our SR semantics preserving implementation of the DoB example consists of 4 tasks: dispatcher, init, and two application tasks. Given the specified sampling periods, the task dispatcher runs at the base period of 1 second.

Figure 9 shows the generated code segments for writing and reading accesses to the shared double buffer.

Figure 10 shows a Simulink model that uses the CTDyB protocol. In this example, the communication buffer trans-

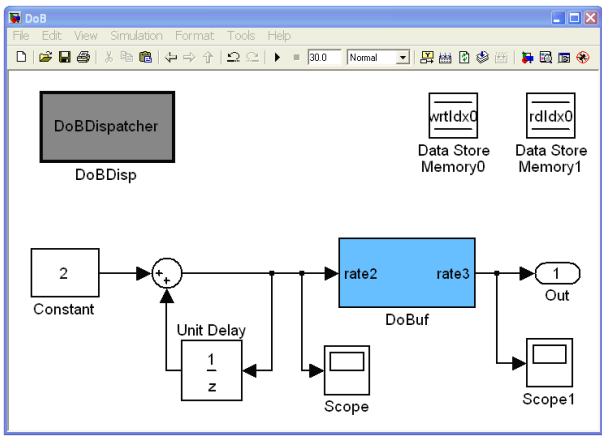

Figure 8. Example of DoB Mechanism

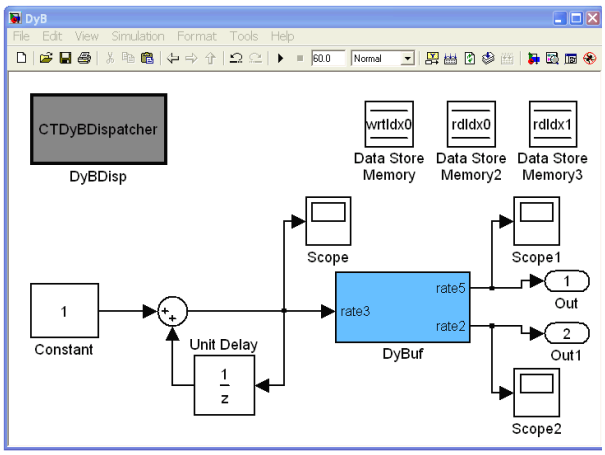

Figure 10. Example of CTDyB Mechanism

fers data between a writer and its two readers. The writer is similar to that in Figure 8 except that the added constant is 1. The sampling periods of the writer, fast reader, and slow reader are assumed to be 3, 2, and 5 seconds, respectively.

Under the Rate Monotonic priority assignment, three buffer slots are needed as per the DyB sizing mechanism. The communication buffer is assumed to take an initial value of 8. Our SR semantics preserving implementation of the DyB example consists of five tasks: dispatcher, init, and three application tasks. The dispatcher is sampled at the base period of 1 second. The generated code for buffer accesses is similar to that of the DoB example.

The code generated by RTW for the SR communication protocol needs to be executed together with an RTOS in an emulation environment. This section presents the experimental environment and the results of the execution of the DyB example. To validate the generated code, the Microchip MPLAB simulation/emulation environment [16] is used. The 40 MHz-10 MIPS PIC18F452 microcontroller from MicroChip Technology is chosen as the target execution hardware platform. The used RTOS is the ePICos18 [17]: a multi-task, preemptive, and real-time kernel that fully conforms to the OSEK OS standard for the PIC18 family. The cycle of the software system timer used to manage alarms and counters in the ePICos18 is 1 msec.

Figure 11 shows the MPLAB SIM results of the generated code from the DyB Simulink model with the CTDyB protocol shown in Figure 10. The solid curve (blue) in Figure 11 is the writer's output versus time (in seconds). The value of the output is incremented by one at its sampling period of 3 seconds. The dashed curve (red) shows the input of reader 1, the slow reader in the DyB example. The simulation results show that reader 1 reads the current output value of the writer, which is expected because the reader has a lower priority than the writer. Similarly, the dotted curve (black) shows the input of reader 2, the fast reader in the DyB example. Because the reader has a higher priority than the writer and the communication link has a unit delay, the first two instances of the fast reader read 8, the initial value of the buffer, before the first instance of the writer finishes its execution. In the following sampling intervals, reader 2 always reads the output value of the writer with a unit delay. The simulation results show that the implemented model executes as expected at run time.

Table 1 shows the RAM consumption due to communication buffers and the ROM consumption of the buffer indexing procedures when implementing the example in Fig-

```
/* Double Buffer Block: '<S2>/S-Function'
 * write side code */
if (rtb_DataStoreRead == 0) {
  DoB_DWork.SFunction_Buffer0 = rtb_UnitDelay;
} else {
  DoB_DWork.SFunction_Buffer1 = rtb_UnitDelay;
}
/* Double Buffer Block: '<S2>/S-Function'
 * read side code */
if (rtb_DataStoreRead1 == 0) {
  rtb_SFunction = DoB_DWork.SFunction_Buffer0;
} else {
  rtb_SFunction = DoB_DWork.SFunction_Buffer1;
}
```

Figure 9. Buffer Write and Read Code

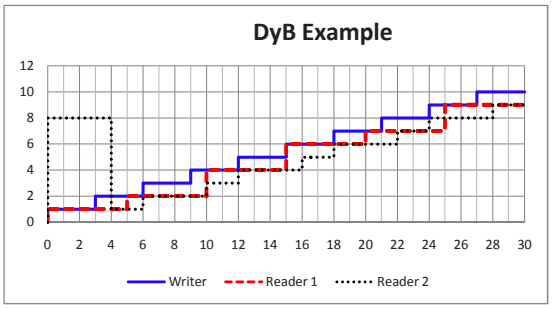

Figure 11. MPLAB SIM Result

		Rate Transition	Double Buffer	Dynamic Buffering
Data Integrity		Yes	Yes	Yes
Time Determinism		No	Yes	Yes
RAM (messages)		4	4	3
ROM (bytes)	Writing	68	64	54
	Reading	58	56	104
	Total	126	120	158

Table 1. RAM and ROM Consumption

ure 10 through the Rate Transition block (only data integrity is obtained), the DoB Protocol, and the CTDyB Protocol. For this particular example, both the RT block and the DoB require 4 message buffers in RAM space, while the DyB only requires 3 buffers. When there are more higher priority readers and the message size is large, the RAM saving can be considerable. Table 1 also shows that the DoB consumes the least amount of ROM while the DyB requires the most amount of ROM for reading and writing procedures.

6 Conclusions and Future Work

We presented the definition of Simulink blocks for the automatic code generation of two implementations of Synchronous Reactive communication channels: DoB and CTDyB. The generated code guarantees both time determinism (stream preservation) and data consistency, for any activation period or activation pattern of the writer and reader blocks. With a moderate increase in ROM consumption, the DyB protocol consumes less RAM compared with the RT block scheme and the DoB protocol. The generated code was validated by analyzing the results of the MPLAB-based emulation on the PIC18F452 processor on which the ePICos18 is running. Code validation confirms that SR communication semantics can be assured upon implementation.

As future work, we may want to take sporadic task activation into account during implementation.

References

[1] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone, "The synchronous languages 12 years later.," in *Proceedings of the IEEE*, pp. 64–83, 2003.

[2] S. Tripakis, C. Pinello, A. Benveniste, A. Sangiovanni-Vincentelli, P. Caspi, and M. D. Natale, "Implementing synchronous models on loosely time triggered architectures," *IEEE Transactions on Computers*, vol. 57, pp. 1300–1314, October 2008.

[3] G. Wang, M. D. Natale, and A. Sangiovanni-Vincentelli, "An OSEK/VDX implementation of synchronous reactive semantics preserving communication protocols," in *Proceesings of the OSPERT*, pp. 38–47, July 2007.

[4] J. Chen and A. Burns, "A three-slot asynchronous reader/writer mechanism for multiprocessor real-time systems," Tech. Rep. YCS 286, University of York, January 1997.

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

[6] 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," *Proceedings of the 5th ACM EMSOFT*, 2005.

[7] C. Sofronis, S. Tripakis, and P. Caspi, "A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling," *Proceedings of the 6th ACM EMSOFT*, October 2006.

[8] J. Chen and A. Burns, "Loop-free asynchronous data sharing in multiprocessor real-time systems based on timing properties," in *Proceedings of the 6th RTCSA*, 1999.

[9] H. Kopetz and J. Reisinger, "The non-blocking write protocol NBW: A solution to a real-time synchronization problem," in *Proceedings of the 14th IEEE RTSS*, December 1993.

[10] M. Baleani, A. Ferrari, L. Mangeruca, and A. Sangiovanni-Vincentelli, "Efficient embedded software design with synchronous models," in *Proceedings of the 5th ACM EMSOFT*, 2005.

[11] M. D. Natale, G. Wang, and A. Sangiovanni-Vincentelli, "Optimizing the implementation of communication in synchronous reactive models," in *Proceedings of the IEEE RTAS*, pp. 169–179, 2008.

[12] *The MathWorks™ Real-Time Workshop®: User's Guide*. http://www.mathworks.com.

[13] "OSEK OS, Version 2.2.3." http://www.osek-vdx.org.

[14] *The MathWorks™ Real-Time Workshop®: Target Language Compiler*. http://www.mathworks.com.

[15] *The MathWorks™ Real-Time Workshop® Embedded Coder™: User's Guide*. http://www.mathworks.com.

[16] *MPLAB IDE, SIMULATOR, EDITOR USER'S GUIDE*. http://www.microchip.com.

[17] G. Wang, "The enhanced PICos18: an O(1) OSEK-compliant real-time operating system," Master's thesis, EECS Department, UC Berkeley, December 2007.