Abstract

Controller area networks (CAN) are widely used in the development of embedded real-time systems. As embedded systems are becoming more complex, the development of dependable software for such systems has become a challenging problem. In this paper, we propose a technique to develop dependable synchronization code for CAN-based embedded systems. Our approach is to factor out synchronization as a separate aspect, synthesize synchronization code and then compose it with the functional code. Specifically, we allow the designer of a CAN-based application to first design core functional code. The designer can then annotate the functional code with control points and specify high-level "global invariants" specifying the synchronization policies. Our methodology generates synchronization code based on message passing in a CAN system and then automatically integrates the synchronization code into the functional code at appropriate control points. We propose and evaluate two solutions: one of which is based on a centralized active monitor and the other is a decentralized solution based on replication. The synchronization code developed is derived from high-level formal specifications via formal methods and is correct-by-construction, and will guarantee high assurance in safety-critical applications.

1 Introduction

The number of real-time embedded electronic systems used in automobiles, tractors, and other systems continues to increase dramatically. These systems typically include subsystems with a large number of devices (sensors and actuators) and separate processors that control the devices. Controller area networks (CAN) have been widely used in design of real-time distributed systems [Fre99] such as automobiles, manufacturing systems, and defense target systems [?]. CAN network is an arbitration based multiaccess network in which arbitration is based on message identities. CAN is a well designed communications bus for sending and receiving short real-time short control messages over shorter distances. Thus, standard scheduling algorithms such as fixed priority preemptive scheduling can be applied to analyze the worst case response time behavior. In this paper, we propose a methodology to synthesize synchronization code for Controller Area Network (CAN) based applications, and we use aspect oriented programming [KLM+97] and object orientation as the core of our development methodology.

Multiple threads in a concurrent program running on various processors must communicate with each other to coordinate their activities. To control such interactions, proper synchronization code must be incorporated at appropriate places in such programs. The development of correct synchronization code for embedded systems is a challenging problem. Reports of mission failure due to implementation errors related to synchronization are not uncommon. The current object oriented design methodologies such as Rational Unified Process [JBR99] does not include a standard methodology for developing synchronization code. As a result, a designer often addresses the synchronization issues along with the development of functional code, which may lead to "code tangling", that is, the synchronization code is tangled with the functional code. The framework of aspect oriented programming deals with this problem by separating the functional code from code for "aspects" [KLM+97]. Following this methodology, the functional code is developed independently and synchronization code is developed as aspect code which is later woven into the functional code to produce complete object-oriented programs. Specifically, our technique is based on the methodology of global invariants advocated in [And91]. We allow a designer to annotate
the functional code with synchronization regions and provide a global invariant that specifies the synchronization to be enforced at the annotated points in the code. Then, a coarse-grain solution is mechanically developed from the global invariant. Finally, we provide a translation of the coarse-grained solution to C code using CAN as the communication medium.

We provide and evaluate two translations into CAN environment. The first one is centralized in nature and is based on the active monitor concept. The second approach involves replicating the control code at each CAN node and utilizes the broadcasting capability of the CAN. The first approach leads to simpler code and less processing at the individual CAN nodes. The second approach, however, uses fewer messages and has lower latency as compared to the first approach and would be preferable in communication intensive applications. We have confirmed our predication via implementation.

Our aspect oriented-based methodology allows easy modification of the synchronization policies. A different synchronization policy can be incorporated by simply supplying a different invariant. The framework will regenerate the new synchronization code and weave it into the functional code. Furthermore, the synchronization code developed using our methodology is “correct-by-construction” and will lead to more dependability in the development of safety critical applications. Although the composite code is guaranteed to satisfy safety properties related to synchronization, it needs to be verified for liveness properties. The S²aVES project at Kansas State University is developing tools for automatic derivation of synchronization code and for automated verification of properties of woven embedded code [SAV01].

2 Overview of the methodology

The overall design methodology is shown in Figure 1. The approach is based on the use-case driven development process of the UML’s Unified Process methodology [JBR99]. The methodology, its integration into UML, and the development of solutions for shared memory systems is discussed in detail in [MSN00]. We first discuss an example to illustrate a typical synchronization problem in an embedded system. Consider a production system shown in Figure 2 which is described in detail in [SRB96]. In this system, the parts arrive on the belt at a certain rate. There are several robots that whose task is to take a part from the belt and pass it to one of the available machines. When the machine is done cutting the part, a robot needs to pick the part from the buffer and place it back on the belt. Each entity is represented by its own agent. This system requires synchronization between agents at several points. First, a part agent needs to communicate (find) with a free robot agent. That is, when a part arrives, we need a free robot as well as a free machine. Once they are available, the robot can pick the part and deposit it in the machine. This presents synchronization corresponding to dynamic group formation and exclusion. For example, since several robots may be free, one of them must be selected to pick the part (mutual exclusion). Similarly, one of the free machines must be selected. Once a robot and a machine has been selected, the corresponding robot agent, the machine agent and the part agent need to form a group (barrier style synchronization). Since the machines share the output buffer, they need to use it in a mutually exclusive manner.

In OS textbooks, synchronization is typically introduced by providing solutions to some abstract synchronization problems such as reader/writers, sleeping barbers and unisex restroom [And91]. The “sleeping barbers” problem is an abstraction of the type of synchronization in the production system discussed above. In this problem, a shop has M barbers, one chair for each barber and a waiting room with K chairs. If all barbers are busy when a customer, say A, arrives, then A waits in the waiting room (provided there is an empty chair). If a barber, say B, is free, then A sits in B’s chair. After B is done cutting the hair, A leaves the shop. Subsequently, B waits for another customer to sit on its chair. As this is a well-known problem, we will use this problem to illustrate the design methodology instead of the production system example. In many applications, however, one may encounter variations of the standard problems (as is the case in the production system example). As a result, solutions to standard synchronization problems may not apply directly to such (ad hoc) problems. Therefore,
designers are often confronted with the task of developing synchronization solutions from scratch. Our methodology is designed to address this by providing a systematic way of generating synchronization code from high-level specifications. The high-level specifications allow variations of the standard problems to be specified and the resulting code generated is guaranteed to be correct.

The following subsections give an overview of the methodology and discuss solutions for CAN-based systems.

- Development of functional code
  The first step in the methodology is the development of the functional code. In this step, all aspects of synchronization are ignored.

- Identifying synchronization regions
  The next step in the methodology is to identify synchronization regions in the functional code. A synchronization region is a segment in a use-case realization (or a code-segment) in which a thread waits for some event to occur or some condition to hold, or in which a thread may trigger an event or change a condition for which another thread is waiting. For example, in the sleeping barber problem, we would annotate the point before the barber cuts the hair (as it is waiting for a customer to arrive) and after it finish the cut (as it waits for the customer to leave). As another example, in the readers/writers problem, the portion of code where a reader (writer) is reading (writing) the shared variables would form a region.

- Invariant specification:
  For each synchronization region, we define two counters (called synchronization counters), an in-counter \( \text{in} \) and an out-counter \( \text{out} \), which are incremented when a thread enters and leaves the region, respectively. Thus, \( \text{in} \) (\( \text{out} \)) counts the number of threads that have entered (exited) the region. The next step in our methodology is to specify synchronization policies via global invariants. A global invariant is defined with connectives \( \land, \lor, \land \) over terms involving the synchronization counters associated with the synchronization regions. For example, if the region associated with customer waiting in the waiting room is called \( \text{wait} \), we would specify the invariant \( \text{in}_{\text{wait}} - \text{out}_{\text{wait}} \geq K \) (number of customer waiting in the waiting room is less than \( K \)). In the readers/writers problem, let \( R \) and \( W \) denote the reader and writer regions respectively. Then, the standard readers/writers policy of allowing at most one writer or any number of readers can be specified as \( ((\text{in}_R = \text{out}_R) \lor (\text{in}_W = \text{out}_W)) \land (\text{in}_W - \text{out}_W \leq 1). \) To allow a variation in which at most \( k \) readers are allowed, we can simply add the conjunct \( \text{in}_R - \text{out}_R \leq k. \) Other variations such as those specifying alternation between readers and writers can be specified using invariants.

- Generation of coarse grained solution:
  The next step is to obtain a coarse-grained solution that increments the in and out counters defined for the regions. Let \( B \) be a Boolean expression (called a guard) and \( S \) be a sequence of statements. The following two types of synchronization constructs are used in a coarse-grained solution[And91]:
  
  1. \( \langle S \rangle \): This statement specifies atomic execution of \( S \).
  2. \( \langle \text{await } B \rightarrow S \rangle \): This statement specifies that the executing process is delayed until \( B \) is true; at which point, \( S \) is executed atomically. No interleaving occurs between the final evaluation of \( B \) and the execution of \( S \).

  The coarse-grain solution for synchronization region \( R \) is obtained as follows: Let \( GI \) be the global invariant for the region \( R \). Let \( ct_R + + \) be an assignment statement associated with synchronization counter \( ct \) for \( R \) (i.e., \( ct \) is either in or out) such that \( ct_R \) is referred to in \( GI \). Then, \( ct_R + + \) is transformed into \( \langle \text{await } B \rightarrow ct_R + + \rangle \), where \( B \) is the weakest condition that satisfies triple \( \langle GI \land B \rangle \). For example, for the client waiting in the waiting room, we will obtain \( \langle \text{await } \text{in}_{\text{wait}} - \text{out}_{\text{wait}} < K \rangle \rightarrow \text{in}_{\text{wait}} + + \rangle \) whereas for exiting from the waiting region, the statement would \( \langle \text{out}_{\text{wait}} + + \rangle \). For the readers/writers problem, the coarse grain solution is:

  **Reader region:**
  
  Entry:
  \( \langle \text{in}_{W} = \text{out}_{W} \rangle \rightarrow \text{in}_{R} + + \rangle \)

  Exit:
  \( \langle \text{out}_{R} + + \rangle \)

  **Writer region:**
  
  Entry:
  \( \langle \text{await}((\text{in}_{R} = \text{out}_{R}) \land (\text{in}_{W} = \text{out}_{W})) \rightarrow \text{in}_{W} + + \rangle \)

  Exit:
  \( \langle \text{out}_{W} + + \rangle \)

- Translation to a solution in CAN environment:
  In this step, we translate the coarse-grain solution to fine-grain synchronization code for the CAN environment. CAN network is an arbitration based multiaccess network in which arbitration is based on message identities. The message to be transmitted among all competing messages is the one with the smallest identity. This makes it possible to encode message priority into an identifier field and to implement priority-based real-time scheduling of messages,
which is essential for embedded systems. CAN is a well-designed communications bus for sending and receiving short real-time short control messages over shorter distances.

Our target environment is a set of C167CR boards connected via a CAN network. Each of the node in the network is a single threaded system. We have developed two translation schemes, a replication-based solution and an active monitor-based solution. The replication-based solution is distributed in nature whereas the active-monitor based solution employs a central controller. For both cases, we present a mechanical way to generate the synchronization code. These two approaches are discussed in details in the next section along with a performance comparison of the two approaches. Both solutions are permission based solution and provide an interface consisting of a method request_permission(region_id, type, src), where region_id is the id of the region, type = 0 denotes permission for entry and type = 1 indicates permission for exit, and src is the id of the source node. We provide the rules to translate the coarse-grain solution into C code. In the resulting code, the nodes communicate via CAN messages to implement the coarse-grain solution.

- **Weaving the code**: The final step in our methodology is that of the task weaver that integrates the component code and the synchronization code. The weaving is simplified in our scheme as the synchronization code provides an interface consisting of the method request_permission. We only need to insert calls to this method at the entry and exit of region in the functional code with the appropriate parameters.

```c
void request_permission(int region_id,
                        int type, int src)
{
    flag = false;
    send_message(region_id, type, src);
    // send a CAN message
    wait until flat is set;
    return;
}
```

After the request message is sent, we wait until flag is set (this flag is set by the Control module when it finds that the permission can be granted). The second function of Order is to impose a total ordering on the request messages (our replication-based is based on total ordering of messages). In a CAN network, a message sent by a node i is delivered to all nodes except the sender, and messages are delivered in the same order to all nodes. However, when a node successfully transmits a message, an interrupt is generated. Using this interrupt, we have implemented a mechanism that enables a node to receive all messages (including its own) in a total order. The ordering module is responsible for receiving messages from the network and assigning sequence numbers to the received messages. At present, we have implemented Order directly on top of basic CAN message passing layer. That is, we have not used any mechanism to guarantee reliability. To prevent overwriting of the network buffer, the priority of the interrupt handler to receive messages is the highest and the amount of processing done in this interrupt handler was carefully kept to a minimum.

The module Control is generated automatically from the set of await statements in the coarse-grain solution. Let the number of regions be m. We assign each region an integer id in the range 0, . . . , m − 1. We use R.id to denote the id of region R. The following rules are used to generate Control:

2.1 Replication-based approach

In this section, we give a systematic approach to generate the code from the coarse-grain solution. The architecture of the replication-based solution is shown in Figure 2. In this solution, the control code is replicated at each node. The code at each site consist of three modules: the application module, control module and the ordering module. We now discuss the generation of code for each of these modules. The application module may request permission to enter or exit a region and the Control modules communicate to coordinate the process of granting permissions. We now gives the rules to generate the C code for each of the modules.

As discussed above, in the functional code (which is the application), we insert a call request_permission(int region_id, inttype, int src) at the entry and exit of each region, where region_id is the id of the region, type = 0 denotes permission for entry and type = 1 indicates permission for exit, and src is the id of the source node. Thus, in the functional code, the execution is blocked until permission is obtained to enter/exit a region.

The ordering module, Order, is independent of the application (that is, the code is independent of the coarse-grain solution). The ordering module performs two functions: The first function is to implement the method request_permission. When this method is invoked, a request message with the parameters region_id, type and src is sent on the CAN. The pseudocode for this method is as follows:

```c
void request_permission(int region_id, int type, int src)
{
    flag = false;
    send_message(region_id, type, src);
    // send a CAN message
    wait until flat is set;
    return;
}
```
We declare two arrays in[m] and out[m], each of size m.

We define a method “boolean CheckCondition(int region_id, int type)”, where region_id is the id of the region and type indicates whether we want to check the condition for entry or exit (type = 0 indicates entry and type = 1 indicates exit). For each region R, we add the following statement to Checkcondition:

If the statement for entry into R is (await(B) → inR +) then add the following statement:

```
if (region_id == R_id) && (type == 0))
    { if B return true else return false
```

If the statement for entry to R is (await(B) → inR +) then we add the following statement:

```
if ((region_id == i) && (type == 0))
    { return true;
```

The statement for exit from R is handled similarly.

The control module processes the messages enqueued in the message queue M_queue created by Order. The pseudocode for processing the messages by the control module is shown in Figure 4. The control module maintains a queue PendQ of request messages whose conditions are currently false. Each message sent over the CAN contains the region id and the type of the request. When a request message is received, we first check the associated condition. If the condition is false, it is enqueued into the PendQ. Otherwise, the corresponding counter is incremented. Furthermore, if the request is from the application at the same site then a permission is sent to the application. As a result of incrementing the counter, it might be the case that the condition for some request in PendQ may become true. Hence, we scan the PendQ for any request whose condition might be true. In order to ensure that all nodes process the messages in the same order, we need to restart the scan each time a request whose condition is true is found.

```
receive request (r,t,src)
    // r = region id, t = type, src = source
    if CheckCondition(r,t)
        then enqueue (r,t,src) in PendQ
    else
        if (t == 0) in[r]++ else out[r]++;
        if src = myid then send permission to application;
        flag = false;
        repeat
            for each request (z,t1,src2) in PendQ
                if Check_Condition(z,t1)
                    then if (t1 == 0) in[z]++ else out[z]++;
                    delete (z,t1,src2) from PendQ;
                    if src2 = myid then
                        send permission to application;
                        break
        until (no request is executed)
```

Figure 4: Message Processing Loop of Control

We have explained above a systematic technique to generate the replication-based solution starting with high-level specifications using invariants. The solution is based on total ordering of messages. Since all control modules receive the request messages in the same order, the order in which the permissions are granted is consistent at different nodes (the total ordering allows us to reach consensus). In the full paper, we show that the solution does indeed satisfy the invariant. We also find that this solution satisfies the concurrent entering property. Informally, concurrent entering implies that if a process p requests entry (or exit) into a region R and the condition for entry is satisfied and remains satisfied, then p will enter R within a bounded number of steps of its own. That is, no other process can delay the entry of p into R. The active-monitor based solution (as well as any lock-based solution) do not satisfy this property because the entry of p could be delayed by another process q that is currently being processed by the active monitor (even though q may be requesting entry to another region). For example, in the readers/writers problem, if several readers request permission concurrently (and no writer is pending) then we would like to grant them permission concurrently. However, in a lock based solution, even though several readers are allowed to be in the reader region concurrently then permissions are granted one-by-one. That is, for each reader, a lock is acquired to get permission and is released after the permission is granted. Subsequently, another reader may acquire to lock to enter reader region concurrently. This solution does not satisfy concurrently because after a reader acquires the lock to obtain permission, it may be delayed for an arbitrary amount of time before it releases the lock. Thus, it is possible for one reader to delay another reader. In the replication-based solution, however, we show that concurrent entering is allowed (e.g., each reader will enter the reader region within bounded number of steps of its own in the above scenario). In the full paper, we formally define this property and show that the replication-based solution does satisfy this property.

The algorithm used in the replication-based solution is based on total ordering of request and is similar to the algorithms in [Lam78, Sch82]. This solution requires one message per request (which is optimal). In Section 3, we show that this solution is more efficient than the active monitor based approach. A drawback of our solution, however, is that since the control code is replicated, nodes may perform extra computation. All nodes receive the same set of request messages and the control modules at different nodes perform identical computation. However, we find that there are several ways in which the amount of computation can be reduced. In particular, if a node p does not have a pending request, then it is not necessary for the control module to process each message according to the algorithm shown in Figure 4. Instead, a node p operates in two modes, busy and idle. The busy mode corresponds to the state when p has a pending request; in this case, it performs the computation as described above. In addition, each busy node also includes
Table 1

<table>
<thead>
<tr>
<th>Delay time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>6</th>
<th>10</th>
<th>20</th>
<th>no load</th>
</tr>
</thead>
<tbody>
<tr>
<td>#customers served (replication approach)</td>
<td>618</td>
<td>748</td>
<td>779</td>
<td>795</td>
<td>807</td>
<td>815</td>
<td>823</td>
<td>831</td>
</tr>
<tr>
<td>#customers served (active monitor approach)</td>
<td>397</td>
<td>504</td>
<td>538</td>
<td>556</td>
<td>569</td>
<td>582</td>
<td>590</td>
<td>601</td>
</tr>
<tr>
<td>% improvement</td>
<td>55.7</td>
<td>48.4</td>
<td>44.8</td>
<td>43</td>
<td>41.8</td>
<td>40</td>
<td>39.5</td>
<td>38.3</td>
</tr>
</tbody>
</table>

call, send_permit, on the proxy. This method sends a CAN message back to the node which requested the permission.

3 Experimentation

Our implementation platform consisted of CR167CR boards connected via a 250Kb/s CAN network. Due to the absence of a real-time OS on the CR167CR boards, we had to implement the entire node as a single threaded application. We have implemented solutions to the sleeping barber problem using the active monitor approach and the replicated monitor approach. We first configured the system with one barber and three clients. The clients arrived after sleeping for a random time in the interval 0..25ms. The barber served a customer for a random time in the interval 0..25ms. A load node was added to the system to adjust the system load. The performance results are shown in Table 1 where we measured the number of customers served by the barber in 30secs. On the x-axis, we varied the delay time, which the time interval between two messages sent by the load node. Under no load, we find that the latency in the case of the replication-based approach is 38% less. As the load increases, the replication-based approach outperforms the active-monitor solution by 55%.

4 Discussion

The approach is flexible in nature.

5 Conclusion

References


on Object-Oriented Programming (ECOOP), LNCS 1241, 1997.


