Modeling the ASCB-D Synchronization Algorithm with SPIN: A Case Study

Nicholas Weininger and Darren Cofer

Honeywell Technology Center, Minneapolis MN 55418, USA
{nicholas.weininger,darren.cofer}@honeywell.com

Abstract. In this paper, we describe our application of SPIN [1] to model an algorithm used to synchronize the clocks of modules that provide periodic real-time communication over a network. We used the SPIN model to check certain performance properties of the system; in particular, we were able to verify that the algorithm achieves synchronization within a time bound, even in the presence of certain types of faults. Our results suggest that state space explosion in models of time-dependent systems can be most effectively managed by explicit modeling of time; by imposing determinism on execution orderings, and justifying that determinism in a domain-specific manner; and by splitting up the space of execution sequences according to initial conditions.

1 Introduction

In this paper, we describe our construction of a formal model of the ASCB-D synchronization algorithm using the SPIN model checker. ASCB-D (Avionics Standard Communications Bus, rev. D) is a bus structure designed for real-time, fault-tolerant periodic communications between Honeywell avionics modules. The algorithm we modeled is used to synchronize the clocks of communicating modules to allow periodic transmission. The ASCB-D synchronization algorithm is a particularly good test case for formal methods, for several reasons:

- The algorithm is a good example of a time-dependent, event-driven system. Many safety-critical embedded software systems, particularly real-time systems, are of this type, and so modeling techniques learned in this effort are likely to have wide application.
- The algorithm is sufficiently complex to test the limits of currently available modeling tools. It demands an intelligent and efficient modeling approach; due to the essentially infinite number of timing sequences possible in the system, a naive model would surely be intractable.
- The central performance properties which the algorithm is intended to fulfill are time bounds. For example, the specification states that synchronization must be achieved within 200 milliseconds of initial node startup. It is notoriously difficult to verify that such bounds hold over all possible startup conditions. Furthermore, these bounds must be shown to hold in the presence of numerous hardware faults, some of which are difficult to simulate on actual test hardware.

SPIN, in turn, proved to be a particularly good tool for modeling such an algorithm; it allowed us not only to verify timing invariants over the state space of the model, but also to conduct random or guided simulations that shed light on the possible behaviors of the model. The graphical representation of these simulations made debugging the model, and eliciting the causes of invariant violations, much easier. Furthermore, the Promela modeling language allowed us to produce an easy-to-understand model that we later found corresponded remarkably well to the C++ implementation code.

Since the ASCB-D synchronization algorithm contains numerous explicit time constraints, one might reasonably ask why we did not choose a model-checking tool that has time-related primitives as part of its language. We did evaluate one such tool, Kronos [2]. In Kronos, a continuously running clock can be associated with each state in the model, and transition conditions between states can include predicates on clock values. However, Kronos's modeling language, based on an explicit representation of states, made for a much less intuitively obvious representation of the algorithm than we obtained with Promela. The high level of algorithmic complexity, in our judgment, made ease of understanding of the representation an overriding consideration.

Despite that complexity, we were able to integrate almost all of the algorithm's features into the model while keeping the state space small enough to allow exhaustive verification of timing properties. We achieved this, first, by minimizing the number of execution orderings possible in the model, and second, by splitting the space of initial conditions and faults into subspaces.

Although the first versions of our model avoided introducing any explicit representation of time, due to our belief that such representation would add unnecessary state to the system model, we found eventually that an explicit time representation was essential to controlling execution orderings. As we incorporated new features into the model, we learned more about the effects of different kinds of nondeterminism on the state space, and were able to gain steadily deeper insights into the behavior of the real system. Indeed, perhaps the most important general lesson we learned was that an iterative, top-down modeling process is crucial to managing model complexity and maximizing understanding of the system being modeled.

The following sections motivate, describe, and examine the results of the modeling effort. Section 2 describes the algorithm we modeled; terminology from this section is used throughout the rest of the paper. Section 3 gives a history of the model's implementation, from simple original versions to the final version, describing in detail the major structural changes we made along the way. Section 4 discusses the lessons we learned.

Our approach built on our experience with real-time operating system modeling using SPIN [3]. Although SPIN and other model-checking tools have been used to analyze real-time, safety-critical systems [4] [5] [6], we have found few cases in which model checking has been used to verify as complex a real-world system as the synchronization algorithm.

2 Algorithm Overview

The ASCB-D synchronization algorithm is run by each of a number of *NICs* (Network Interface Cards) which communicate via a series of buses. These NICs are split into two *sides*, corresponding to the pilot and co-pilot sides of an aircraft. For each side there are two buses, a primary and a backup bus. Each NIC can listen to, and transmit messages on, both of the buses on its own side. It can also listen to, but not transmit on, the primary bus on the other side. From the viewpoint of a given NIC, other NICs on the same side are called *onside* NICs; NICs on the other side are *xside* NICs. Likewise, the buses on the NIC's own side and on the other side are onside and xside buses respectively. The basic structure of buses and NICs is diagrammed in Figure 1.

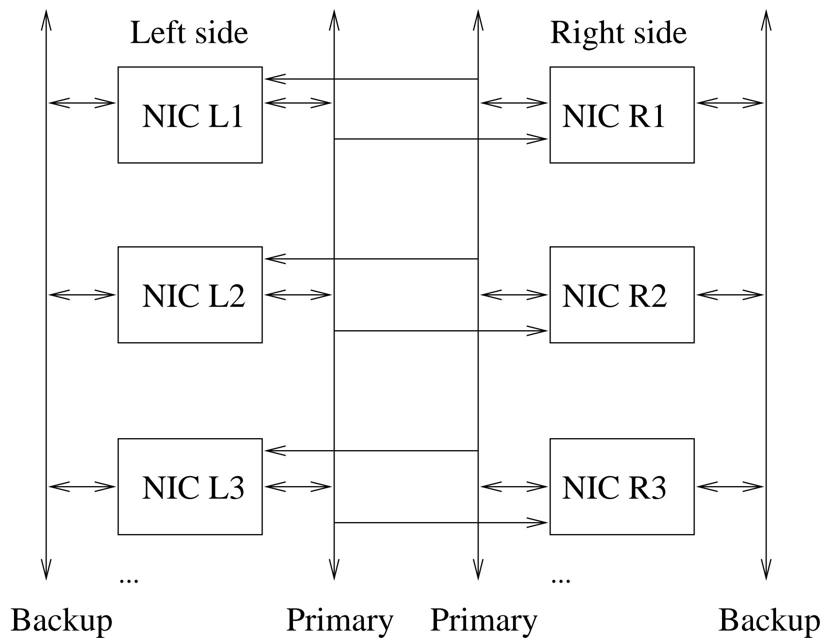

Fig. 1. Structure of the ASCB-D buses.

The operating system running on the NICs produces *frame ticks* every 12.5 msec which trigger threads to run. In order for periodic communication to operate, all NICs' frame ticks must be synchronized within a certain tolerance. The purpose of the synchronization algorithm is to enable that synchronization to occur and to be maintained, within certain performance bounds, over a wide range of faulty and non-faulty system conditions.

The synchronization algorithm works by transmitting special *timing messages* between the NICs. Upon initial startup, these messages are used to desig-

nate the clock of one NIC as a “reference” to which the other NICs synchronize; after synchronization is achieved, the messages are used to maintain synchronization by correcting for the NICs’ clock drift relative to each other. The algorithm is required to achieve synchronization within 200 msec of initial startup. It must do this regardless of the order in which the NICs start or the time elapsed between their initial startup.

The synchronization algorithm must also meet the 200 msec deadline in the presence of malfunctions in certain NICs or buses. For example, any one of the NICs might be unable to transmit on, or unable to listen to, one or more of the buses; or it might babble on one of the buses, sending gibberish which prevents other messages from being transmitted; or one of the buses might fail completely at startup, then function again for some period of time, then fail again, then function again, and so on.

3 Implementing the model

Our implementation of the ASCB-D synchronization algorithm in a SPIN model aimed to begin by modeling the smallest possible nontrivial subset of the specified algorithm, and increase the complexity of the model in stages from there. We therefore began by limiting the scope of the model to the initial “synchronization establishment” phase of the algorithm, omitting the logic dealing with maintaining synchronization after it has been achieved. We initially modeled only two NICs, one on each side. Also, we decided to abstract away from the details of message transmission on the bus, and in general to make the assumption that both transmitting messages and executing code take zero time.

We also initially decided to abstract away from the numerical time calculations used by the algorithm, and to model only the ordering constraints it imposes. This decision was based on a strong initial aversion to the idea of an explicit model of time. Since the state space of a model must be finite, elapsed time has to be measured in discrete units of some fixed granularity; no matter what granularity is chosen, this strategy is prone to errors caused by lumping two different execution orderings together into the same time unit. Furthermore, since time counters can typically take on a very large number of different values, use of time counters can greatly increase the model state space size. The explicit modeling of time eventually turned out to be unavoidable, and we were forced to find ways around its limitations.

3.1 Simplifying observations

Beyond our initial restrictive assumptions, we can make two observations about the structure of the ASCB-D algorithm which limit the space of execution orderings which we must consider.

First, from a single NIC’s standpoint, the algorithm can be viewed as a process driven by the frame tick event. Each time the NIC receives a frame tick, it executes some code and then waits for the next tick. Furthermore, the code

executed after a frame tick operates entirely on input data that were collected before the frame tick.

This means that if any two NICs, NIC 1 and NIC 2, are executing at the same time, the execution sequence of NIC 1 cannot depend on anything that NIC 2 does while NIC 1 is executing. Therefore we can construct the model such that iterations of the algorithm on different NICs are atomic with respect to each other. This greatly reduces the size of the state space by eliminating interleavings between different NICs' execution.

Second, the frame ticks of different NICs are related in their periodicity. It is not possible for NIC 1 to go through an arbitrarily large number of frame ticks while NIC 2 gets none, so execution sequences in which this happens should be excluded from the model. In fact, under most circumstances, NIC 1 can only have one frame tick for each tick of NIC 2, and vice versa.

The key phrase here, however, is "under most circumstances." Because frames on different NICs *can* be of different lengths during the initial establishment of synchronization, there are legitimate execution orderings in which NIC 1 gets two ticks to NIC 2's one. The need to include these orderings in the model, without including unrealistic n-to-1 orderings, was the key factor that eventually drove us to implement an explicit numerical representation of time.

3.2 Structure of the two-NIC model

In our first and simplest model, the two NIC processes wait for ticks provided by a "time" process; these ticks are communicated through blocking, or rendezvous, channels. When a NIC receives a frame tick event, it queries a buffer process which gives it the contents of the timing messages received in the previous frame. The NIC executes the algorithm logic appropriately, based on the messages received from the buffer process. It then tells the buffer process, again using a rendezvous channel, what (if any) timing message to send in the coming frame. The communications occurring among the processes are diagrammed in Figure 2.

The NIC processes' code is enclosed in atomic statements specifying that they are not to be interleaved with each other. Because the buffer processes are distinct from the NIC processes, however, their message transmissions may be interleaved with NIC executions. We separated buffers from NICs in order to model execution orderings in which messages are sent after a frame tick, and to prevent deadlocks which could occur if each NIC process waited for the other to send a message.

However, in the two-NIC model, the separate buffer processes did not solve the deadlock problem, but only pushed it one step further out. The deadlock sequence instead became one in which NIC 1 was waiting to send to Buffer 1, which was waiting to send to Buffer 2, which was waiting to send to Buffer 1 (and likewise NIC 2 was waiting to send to Buffer 2). Also, introducing separate buffers led to too many spurious execution orderings in which a message was sent to a buffer at the frame tick, but then not sent on to the other NIC until after that NIC had undergone several frame ticks.

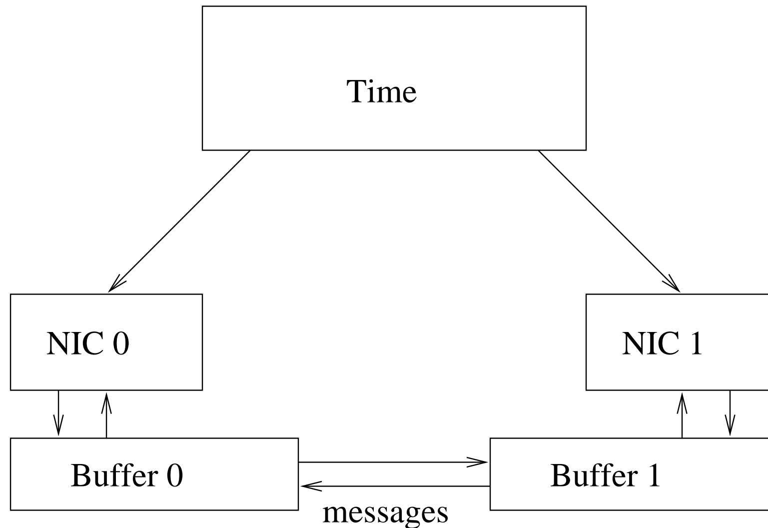

Fig. 2. Processes in the first two-NIC version of our model.

A solution to this problem is to have NICs modify global data structures in order to send messages, rather than making them communicate with separate buffer processes. This approach eliminates deadlocks, since there is no interprocess synchronization mechanism required. However, it tends to expand the state space, since global data structures are less susceptible to SPIN's compression algorithms.

Despite these limitations, the two-NIC model is useful. The logic modeled is very far from the complexity of the real algorithm, but it nevertheless encodes the basic strategy. The model is also sufficient to encode and verify a key system invariant: that no matter what order the two NICs start up in, they always eventually get in sync. In this model, that invariant can easily be encoded as an LTL property and exhaustively verified. By introducing a counter that is incremented each tick after both NICs have started up, we can also bound the time required for both NICs to get in sync.

An LTL verification, with global data structures instead of buffers, that the NICs are both in sync within 12 frames of starting completes in about 15 seconds and visits 150409 states, reaching a search depth of 31741. (All timing figures are for a 300 Mhz UltraSparc II with 1024 MB of RAM). This result led us to predict that modifications would be needed to keep the state space size tractable with a more complex four-NIC model.

3.3 Expanding the model to four NICs

Our initial four-NIC model incorporated the following changes:

- We extended the time process straightforwardly to provide ticks to four NICs. Each time that the time process "ticked," it would allow for one frame tick for each NIC; however, it did not restrict the order of the frame ticks corresponding to its tick. This created problems, as we discuss below.
- SPIN verifies LTL properties by generating verifier processes which run concurrently with the model's processes. This increases the state space size considerably. We replaced the LTL property by a simple assertion within the time process:

  ```
  assert(frames_since_started < 9 || (in_sync[0] && in_sync[1]
  && in_sync[2] && in_sync[3]));
  ```

  This assertion is considerably easier for SPIN to verify. It is checked only once per frame tick, but since the variables tested only change once per frame tick, this is sufficient.
- We reintroduced buffer processes, but with a separation of onside and xside buses. Since each NIC sends messages only on its onside bus, but listens to both onside and xside buses, this prevents the buffer processes from deadlocking. We also introduced logic in the algorithm that differentiates between onside and xside timing messages.

Our changes produced a stubbornly intractable model. Even after making further simplifications (e.g. disallowing all those execution sequences where one NIC started up two or more frames before the others), we estimated (by running supertrace verifications) that the state space size was in the tens of millions, and the search depth exceeded 500000. This goes far beyond the capacity of the hardware available to us.

Furthermore, our initial four-NIC model clearly included execution sequences not possible in the real system, because of the fact that the time process did not control the ordering of the NIC ticks corresponding to its tick. For example, if we number the NICs 0 through 3, we can see that the time process would allow ticks and thus message transmissions to occur in a 3-2-1-0-0-1-2-3-3-2-1-0-0-1-2-3 sequence, which is not possible in the real algorithm. However, imposing a fixed order on NIC ticks goes to the opposite extreme, excluding many orderings which the real algorithm does allow. For instance, if a NIC extends its frame length to "sync up" with the others, there could be two frame ticks from another NIC arriving within that NIC's frame.

3.4 The time/environment process

The problems in the four-NIC model clearly necessitated major changes to the basic model structure. The introduction of an explicit numerical time model, and the combination of that time-modeling capability and the message-transmission capability in the same "environment" process, turned out to be the changes we needed to make the model tractable again.

For our first time model, we chose a granularity of 0.5 msec. This allowed us to capture many of delay times in the algorithm, and to represent most time quantities of interest (e.g. the length of frames) within an unsigned byte, minimizing the total number of bytes that must be added to the state space. We later reduced the time granularity to 1 μsec in order to capture more of the delay times precisely; the sufficiency of a 1 μsec granularity is justified in Section 3.8.

The time/environment process keeps track of the time remaining until the next frame tick of each NIC and the messages received by each NIC in the current frame, as well as the total time elapsed since "time zero." It then sits in a loop executing the following algorithm:

```
while(forever)
{
     pick id such that timeToNextTick[id] is minimal;

     send NIC[id] the contents of its message buffers from
     the last frame;

     wait for NIC[id] to send back the length of its next frame,
     plus the contents of the message it wants to send;

     if that message is not empty, send it to the other
     NICs' buffers;

     add timeToNextTick[id] to timeToNextTick[i] for all i != id;

     add timeToNextTick[id] to total_elapsed_time;

     set timeToNextTick[id] to the length of the next frame
     for NIC[id];
}
```

Observe that the above algorithm transfers the responsibility for determining the length of the next frame to the NIC, allowing the introduction of NIC-specific algorithm logic for determining this length. Also, it encapsulates message transmissions in the time process, eliminating the need for separate buffer processes. Since the time process is now dispatching the NICs one by one, there is now no fear of message transmission deadlocks.

The NIC process can then sit in the following loop:

```
while(forever)
{
     wait for the time process to send the contents of the
     message buffers from last frame;

     process the message buffers appropriately depending
     on the NIC's current state;

     compute the length of the next frame and the contents
     of the timing message to send (if any);

     send these data to the time process;
}
```

The communication structure between NIC and environment processes is diagrammed in Figure 3.

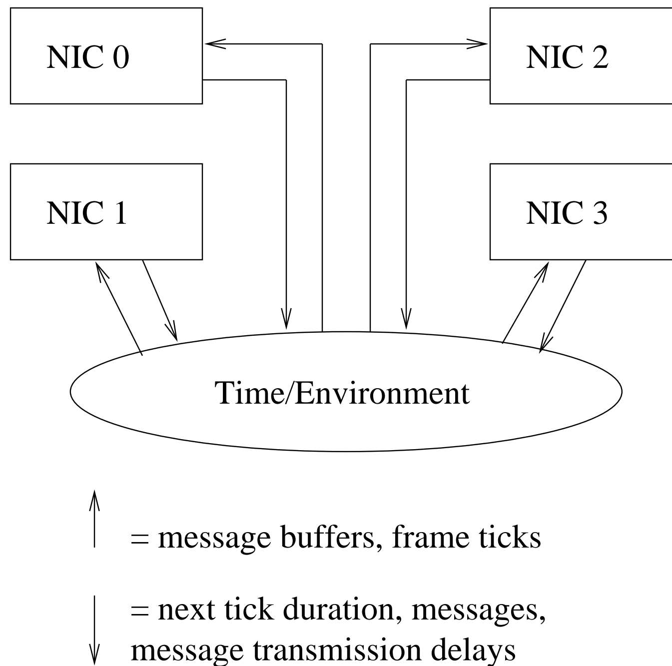

Fig. 3. Communication between four NICs and the environment process.

3.5 Consequences of the time/environment process

A number of fundamental changes in the model follow immediately from this introduction of numerical time. These include:

- The environment process neatly encapsulates all those parts of the system that provide input to the algorithm we wish to model (frame ticks, buffers, and buses), while the NIC process encapsulates that algorithm completely. The interface between the two is simple and localized. As we shall see later, this is perhaps the most powerful advantage of the time/environment model; it allows faults to be injected and complicated hardware interactions to be added with no change required to the NIC code.
- Because the environment process now dispatches ticks one by one, NICs are trivially guaranteed to execute atomically with respect to each other. NICs also execute atomically with respect to the environment process. This simplifies the space of possible execution orderings dramatically; the only order that matters is the order in which ticks and message transmissions occur.
- Complicated tick orderings produced by frames of different lengths are now explicitly and accurately represented in the model.
- We can now easily test for timing-dependent system properties. For instance, we can place an assertion in the environment process, checked before each tick, that states that all NICs should be in sync within 200 msec of the startup time:

```
assert((total_time_elapsed < 400) ||
(in_sync[0] && in_sync[1] && in_sync[2] && in_sync[3]));
```

- Because the interface between environment and NIC includes all the data that must be shared between them, there is no need for global data structures. This allows SPIN's compression techniques to reduce the memory required to store each state.

The reduction in state space produced by eliminating extraneous buffer processes and impossible interleavings far exceeds the increase produced by having time counter values. However, we still face the problem that letting total time elapsed go off to infinity can expand state space unnecessarily. To solve this problem, we observe that the timing invariants we want to verify in the system provide a natural bound on the maximum "interesting" time value. If we require that all NICs get into sync within 200 msec, then there is no reason to continue searching the state space after $200 + \epsilon$ msec has gone by. Therefore we can put a test in the time process that, when a certain time threshold is passed, places the system in an "acceptance" or valid end state.

Two major issues remain. First, in the above loop description, we stated that the environment process should "choose `id` such that `timeToNextTick[id]` is

minimal." We did not specify how to choose between two NICs whose ticks are scheduled at the same time: should we make a nondeterministic choice, or put them in some predetermined order? The former will greatly expand the state space; the latter runs the risk of excluding important execution orderings. In our initial model, we opted to choose nondeterministically; further expansion of the model eventually forced us to change to deterministic choice, and to justify that change.

Second, initial startup orderings can now be specified by giving initial times (i.e. times to first tick) for each NIC. This is an important new capability, because we can now ask "what if the NICs started at just these times, and ran from there?" However, it also means that much of the nondeterminism from the original model has now been transferred into the choice of initial starting times. Effectively, we have split the state space of all possible execution sequences into many small slices, one for each possible set of initial conditions. This makes verification runs tractable, but it also forces us to be careful about what we deduce from an exhaustive verification: before we conclude that a timing property always holds in the system, we must verify it for a sufficiently representative set of initial conditions.

We can, of course, introduce nondeterministic choice of initial conditions into the model. Both the initial startup times and the order in which the NICs are assigned to those times can be chosen "randomly." However, choosing randomly over even a small range of possible times quickly expands the state space beyond tractability. One version of our four-NIC model, for example, can verify our 200 msec time-to-sync bound for a fixed set of startup times in about six minutes, storing 3.05 million states and reaching a search depth of 2113. Here we use nondeterministic NIC ordering for that set of startup times, so we are verifying over 24 sets of initial conditions. If we also chose the startup time for each NIC randomly from among a set of five values, we'd have 625 * 24 sets of initial conditions over which to verify; this would likely produce a state space far too large for a single verification run.

3.6 Adding delayed message transmission

Once we model time explicitly, we can introduce into the model the variable delays that occur between frame ticks and timing message transmissions. Since we are modeling the hardware abstractly with the environment process, the approach is simple: let the NIC process send the message delay to the environment process, and have the environment process schedule the message transmission as an event at the appropriate time.

This requires that we add four new events to the list from which the environment must select the "soonest," since each NIC might transmit a timing message each frame. If nondeterministic choices are allowed, this will increase the state space size considerably. We therefore specify that message transmissions always occur after frame ticks. This requires some justification to ensure that real execution orderings are not eliminated from the model; see our more general argument in Section 3.8.

3.7 Adding multiple buses and bus fault injection

Once we've made the unified environment process into a single entity distinct from the NIC processes, it is easy to implement a model of the multiple buses (primary and backup on each side) that exist in the ASCB-D system. The actual transmission logic is handled entirely within the environment process. Modeling multiple buses also provides an opportunity to start injecting bus faults into the model. Each NIC might be deaf (unable to receive messages) with respect to the onside primary bus, the onside backup bus, and/or the xside primary bus; it might also be mute (unable to send messages) on either onside bus.

An example of the relevant logic is shown below. Here what we are seeing is the code that sends a message from one timing NIC to the other timing NIC on the same side. The sending NIC transmits the message on both primary and backup buses; the receiving NIC listens to only one of those buses at a time. In order for the receiving NIC to get the message successfully, the following must be true for one of the buses, either primary or backup:

1. The receiving NIC must be listening to that bus.
2. The receiving NIC must not be deaf to that bus.
3. The sending NIC must not be mute to that bus.

If these conditions are satisfied for one of the buses, the message is placed in the receiving NIC's first onside buffer, and the receiving NIC's second onside buffer is filled with the former contents of its first onside buffer.

```
if
   ::(listening_primary[(enabled_id + 1) % MAX_NICS] &&
     !onside_primary_deaf[(enabled_id + 1) % MAX_NICS] &&
     !primary_mute[enabled_id]) ||
     (!listening_primary[(enabled_id + 1) % MAX_NICS] &&
     !onside_backup_deaf[(enabled_id + 1) % MAX_NICS] &&
     !backup_mute[enabled_id]) ->
     d_step {
       buffers[(enabled_id + 1) % MAX_NICS].onside_1 =
buffers[(enabled_id + 1) % MAX_NICS].onside_0;

       buffers[(enabled_id + 1) % MAX_NICS].onside_timestamp[1] =
        buffers[(enabled_id + 1) % MAX_NICS].onside_timestamp[0];

       buffers[(enabled_id + 1) % MAX_NICS].onside_0 =
last_message[enabled_id];

       buffers[(enabled_id + 1) % MAX_NICS].onside_timestamp[0] =
        timeToNextTick[(enabled_id + 1) % MAX_NICS];
     }
   ::else ->
fi;
```

Now that we have the ability to specify fault conditions, there are several ways to inject faults. The simplest way is to set some combination of fault attributes true at the beginning of the environment process, before any events have occurred; this makes the fault specification part of the initial condition specification. For example, if we wished to model a disabling of the left-side primary bus on powerup, we could set:

```
onside_primary_deaf[0] = true;
onside_primary_deaf[1] = true;
xside_deaf[2] = true;
xside_deaf[3] = true;
```

We might also wish to model a fault that occurs as soon as a certain condition becomes true (e.g. a certain amount of time has elapsed). To do this, we check the condition after every tick or message event processed, and call a fault injection routine when it becomes true.

```
if
::!fault_injected ->
     if
     ::condition ->
         inject_fault();
       fault_injected = true;
     ::else -> skip;
     fi;
::else -> skip;
fi;
```

Furthermore, we might like to model a fault that is injected at a random time; for instance, we might want to model a bus that could malfunction at any time. This can be done as follows:

```
if
::!fault_injected ->
     if
     ::condition ->
         inject_fault();
       fault_injected = true;
     ::true -> skip;
     fi;
::else -> skip;
fi;
```

In these examples, a fault event is something that can occur only once in any execution sequence of the model. However, we might also like to model faults that occur multiple times; a bus, for example, might switch from functional to nonfunctional many times during execution. We can do that by getting rid of the

"fault_injected" variable above and modifying the inject_fault() function so that it "uninjects" a fault already injected. For example, an inject_fault() function designed to make the onside primary bus malfunction might do this:

```
if
::onside_primary_deaf[0] ->
     onside_primary_deaf[0] = false;
     onside_primary_deaf[1] = false;
     xside_deaf[2] = false;
     xside_deaf[3] = false;
::else ->
     onside_primary_deaf[0] = true;
     onside_primary_deaf[1] = true;
     xside_deaf[2] = true;
     xside_deaf[3] = true;
fi;
```

Note, however, that nondeterministically choosing to execute such a fault function will greatly increase the state space size, much more than if the fault can only occur once. We found that introducing a fault condition that could "flip" arbitrarily many times, as in the above example, made the state space too large to be tractable for exhaustive verification.

3.8 Adding sync phase behavior to the model

With the addition of multiple buses and bus faults, our model now incorporated most of the features of the algorithm relevant to the initial achievement of synchronization. However, most of the complex logic in the ASCB-D synchronization algorithm, especially logic for detecting and responding to faults, is specific to NICs in the "already synchronized" state. In principle, adding this logic to the model should be a relatively simple matter of extending the NIC processes to do more complex calculations on messages received when in sync; after all, the environment process cares only about frame ticks and message transmissions, and not about whether NICs are in sync.

However, our initial experiments quickly showed that the existing environment process, combined with sync phase code, produced intractably large models. The culprit here was clearly the nondeterministic choice between events that were scheduled to happen at the same time. The defining characteristic of sync phase is that all NICs in sync should have their frame ticks occur at the same time. Furthermore, corresponding pairs of NICs (one on each side) transmit timing messages at the same time when in sync; for example, NIC 1 and NIC 3 transmit at the same time. Thus a four-NIC model in normal operation with all NICs synced must explore 96 orderings of the NICs' frame ticks and subsequent sync message transmissions– for *every frame.*

In order to make the model tractable again, we had to impose a deterministic priority ordering on events scheduled at the same time. Intuitively, we ought to

be able to impose such an ordering and then replicate the execution orderings lost by changing initial conditions. For example, if two timing messages from two different NICs, say A and B, are scheduled at the same time with a certain set of startup times for each NIC, they might come in either order. If we impose an order, say A comes before B, then we can replicate the "lost" execution sequence where B comes before A by making B's startup time slightly earlier, so that the time at which B is scheduled to transmit is strictly earlier than the time A is scheduled to transmit. This works as long as the time increment by which we change B's time is small enough that it will not affect any other event in the system. In effect, we are splitting the old state space into two subspaces, each of which can be covered by a (hopefully) more tractable verification.

We want to formally justify the imposition of an order rule in which:

- when a frame tick and a message transmission are scheduled at the same time, the frame tick comes first.
- when two message transmissions are scheduled at the same time, either could come first.

Our argument can be stated as follows: Suppose there are two events, E_A and E_B, which emanate from NICs A and B respectively, and which in a certain system state S are scheduled at the same time. If such events are nondeterministically ordered, we could have E_A or E_B execute first, leading to consequent states C_1 and C_2. Given an order rule for such events satisfying the above properties, we want to show that there exist states S_A and S_B using that order rule, such that the consequent states of S_A and S_B, C_A and C_B, are equivalent to S_1 and S_2 respectively. Here "equivalent" is defined as "differing only in the values of time counter variables." If this is true, then we've proven that any execution ordering in the nondeterministic model is also in the deterministic model.
We proceed by cases.

1. E_A and E_B are both tick events.
   Then $C_1 = C_2$, so we can take $S_A = S_B = S$. This is true because NIC A's processing of its frame tick cannot in itself affect NIC B in any way, and vice versa; furthermore, their resultant timing message transmissions (if any) will be scheduled at just the same times regardless of the order they execute in. Therefore the ordering of E_A and E_B cannot affect future events.
2. E_A is a tick event and E_B is a message transmission, and $A = B$.
   Then we make the assumption that a NIC's message delay time (the time between the occurrence of the frame tick and the transmission of the timing message resulting from that frame tick) is always either:
   - strictly less than the length of its next frame
   - infinity (i.e. the NIC transmits no message from that frame tick)

   This assumption is true for all versions of our model. It implies that if E_A is scheduled at the same time as E_B, then E_B must be the message transmission resulting from E_A. If this is the case, then E_A must execute before E_B, since E_B does not exist until E_A executes. So, whatever order rule we choose, there is only one ordering of these two events.

3. E_A is a tick event and E_B is a message transmission, and $A \neq B$; or E_A and E_B are both message transmissions.
   For the first of these, under our deterministic order rule, we can take $S_A = S$ since E_A is a tick event and so must execute first. For the second, we can again assume that our order rule puts E_A before E_B, so $S_A = S$. In both of these cases, we want to construct S_B by setting an initial state for the system that is the same as the initial state resulting in S, except that the startup time for A is one time unit later.
   If A or B is in sync, then there are only two ways E_A and E_B can occur at the same time:
   (a) They are message transmissions from two NICs on different sides. In this case, their order is unimportant, since they go into different buffers for each NIC. We therefore can take $S_B = S$ if this happens.
   (b) One of A and B is not in sync with the other. If this is true, we can move the startup time for B back one unit so that E_B occurs before E_A. This works as long as all of the other events before E_B and E_A are not affected by this shift.

   In this case, then, we must ensure that the time granularity is sufficiently small that we can move B's events by one unit without affecting the orderings of those events with respect to others. Once again, the ordering of two ticks is irrelevant, and we can without loss of generality assume that B is out of sync with respect to at least one other NIC. Then it is sufficient to make the time granularity no greater than $D_{min}/(n * F_{max})$, where D_{min} is the shortest nonzero message transmission delay for a non-synced NIC, n is the number of timing NICs, and F_{max} is the maximum number of frames that a NIC can take to get in sync.
   As it happens, for our model, $D_{min} = 500$ μsec, $n = 4$, and $F_{max} \leq 20$. This gives a maximum time granularity of 6.25 μsec, so the 1 μsec time granularity used in the final version of our model is sufficient.

3.9 Including fault response logic

Once we have a tractable model of normal sync operation, we can then model the rules by which NICs detect and respond to bus faults. These situations can be tested in the model by the introduction of appropriate bus faults. For instance, if the left-side primary bus malfunctions after the NICs have achieved sync with a left-side NIC as reference point, the right-side NICs will no longer have a reference point for maintaining synchronization. Eventually one of them will assume the status of a new reference point, and the left-side NICs will have to resynchronize to it.

In the presence of such resynchronization, verifying the timing invariant becomes more complicated. Now it is clearly not enough to look at the first $200 + \epsilon$ msec of operation. Besides verifying that we initially get into sync within 200 msec, we want to verify that when sync is broken by some fault, resynchronization is achieved within some bounded time. We need two modifications to our time-measurement mechanisms in order to make this work:

1. We introduce, in place of a counter for total elapsed time, a counter variable "total_time_since_resync" that measures the total time since the last resynchronization began. This counter variable is set to 0 at startup, incremented just as the elapsed time variable was before, but reset to 0 upon the occurrence of a resync (i.e. all NICs are in sync, then one NIC drops out of sync).
2. We now cannot cut off the state space when total_time_since_resync passes a certain threshold; there is no obvious bound on how long the system might stay in sync and keep generating new and "interesting" execution orderings. Therefore, when total_time_since_resync exceeds its threshold, instead of jumping to an end state, we set total_time_since_resync back to the threshold value. This prevents it from increasing to infinity, and bounds the state space by causing SPIN to treat two states as the same if they differ only in the amount of time elapsed since resynchronization.

We found that the "in-sync" model remains tractable, given appropriate restrictions. For instance, a verification run for our timing invariant, with a fixed set of startup timings, plus a randomly occurring bus fault, finishes in about five minutes after exploring 2.65 million states, reaching a search depth of 42724. If a totally deterministic startup ordering (all NICs start at the same time) is used, and there is no bus fault, the verification completes in less than a minute, storing 65032 states and reaching a depth of 9472.

Note that this verification run proves less than the same run would have for a version of the model with nondeterministic event ordering. However, this can be remedied by making several verification runs with different sets of startup timings. The important point is that we have kept the model tractable while greatly increasing its complexity, decreasing the time granularity, etc.

3.10 Expanding the model

As a final test of our model's scalability, we expanded it to include six rather than four NICs. The changes required to the environment process were relatively simple: We added two new events to the event list for the frame ticks of the two new NICs, expanded the appropriate arrays (message buffers, times remaining until the next tick, etc.) to account for the new processes, and expanded the message transmission code to transmit messages to the new NICs.

When we ran an exhaustive verification with six NICs, for a fixed startup ordering and no faults, the run again completed in less than a minute, storing 92951 states and reaching a depth of 13754. This was much less of an increase in state space size than we had feared; it served to confirm the effectiveness of our environment process design, which prohibits interleavings between the user NIC code and the other processes' code. Adding a random bus fault increased the time required to about nine minutes, the state space to 2.65 million states and the search depth to 59050.

However, adding a nondeterministic choice of startup order, with no faults, resulted in the verification using 2 million states, reaching a depth of 14805

and taking 5 minutes to complete. With a randomly occurring bus fault and nondeterministic startup order, the model becomes intractable for exhaustive verification; bitstate verification shows an approximate size of 16 million states. The reason for this is easy to see: whereas with four NICs there were 24 possible startup orderings for any fixed set of startup times, with six NICs there are 720. A six-NIC model, then, is pushing the limits of verifiability with our current computing resources. (The actual system in which the algorithm is used can contain up to 32 NICs, but the NICs are organized into categories such that six are sufficient to represent all possible interactions).

Fortunately, the model's usefulness does not lie only in its capacity for verifying system properties. Our current model, which integrates a large majority of the specified algorithm's features, is a very powerful tool for asking "What if?" questions that are difficult to answer either by manual analysis or by testing on real hardware. If you want to see what happens to the model when, say, the NICs start out at intervals of 0.2 msec, and the right-side primary bus fails 40 msec after the first NIC begins, all you need to do is program in (mostly with #define statements) the appropriate startup timings and fault injection conditions, and run a simulation. The resultant graphical display, although it can be hard to read at first, gives a clear and comprehensive picture of what's happening in the system at any point in time.

4 Lessons learned

4.1 Efficacy of the top-down modeling approach

We believe that the success of our modeling effort– our construction of a working, useful, well-understood model of a complex algorithm– demonstrates the efficacy of what we might call a "top-down" modeling process. The crucial characteristic of such a process is that it starts by modeling as small a subset of the system as possible at the highest possible level of abstraction, and builds on that small model in stages, gradually decreasing the level of abstraction and increasing the number of features included. This approach may seem to involve a lot of extra work, but in fact it reduces the model construction time required significantly; most of our model was built by one person with about 1.5 man-months of effort.

There are several reasons why the top-down approach works well. It avoids the conceptual difficulty of trying to comprehend and model a lot of unfamiliar things at the same time; the modelers' understanding of the system is built up in stages as the model itself is built. Because the top-down approach starts with a highly abstracted model and decreases the level of abstraction as time goes on, it facilitates high-level thinking and prevents too much focus on the trees at the expense of the forest. The approach drives the model structure naturally toward modularity and clear separation of functions, leading to an easier-to-understand model. Furthermore, even partial models can provide important insights into a complex specification, and can form the basis for thinking about features of the system that are not yet modeled.

4.2 Modeling time-dependent systems

The ASCB-D modeling effort also taught us quite a bit about modeling systems whose central properties are based on time. Modeling these systems with a purely order-based model introduces large numbers of execution interleavings which do not exist in the real system and which produce spurious violations of safety properties. Furthermore, performance requirements for these algorithms are often expressed in terms of time, so you need a numerical time model to verify them.

Modeling time with an event queue seems to be a feasible, conceptually simple method for event-driven algorithms. However, it requires that careful thought be given to nondeterminism in simultaneously scheduled events. Some restrictions on nondeterminism may have to be justified by algorithm-specific system properties, as we justified the restrictions on event ordering.

4.3 Controlling model state space size

Finally, our experience taught us that controlling nondeterminism and interleavings of concurrent processes is the overriding factor in managing the state space of a model– far more important than limiting the number or size of variables used, reducing the total number of processes in the model, etc. It is useful to assign a single process to be the manager of interleavings, so that other processes can model the desired algorithms in a way that is close to how they would be implemented in the real system. The construction of this process is likely to be less susceptible to automation than the modeling of the algorithm itself, since the structure of this "environment" depends on domain-specific properties of the hardware events that drive the algorithm.

One of the most effective tactics for controlling nondeterminism is using fixed initial condition settings to split the state space of all possible system conditions into numerous subspaces. When attempting to verify an invariant with such a model, it is always necessary to qualify the verification result by noting the set of initial conditions over which the verification was performed. Complex, time-dependent systems are likely to have a space of initial conditions large enough that the system cannot be verified over the whole space in a single run; therefore small subsets of the initial condition space must be chosen intelligently as subjects for verification. The formulation of a small set of initial conditions that are in some sense representative of all the possible ones is a difficult and domain-specific problem, which we have yet to address for the ASCB-D synchronization algorithm.

References

1. G. Holzmann. The SPIN Model Checker. *IEEE Transactions on Software Engineering*, vol. 23, no. 5, May 1997, pp. 279-295.
2. S. Yovine. "Kronos: A verification tool for real-time systems." In *International Journal of Software Tools for Technology Transfer*, vol. 1, no. 1/2, Oct. 1997.

3. J. Penix, W. Visser, E. Engstrom, A. Larson, and N. Weininger. Verification of time partitioning in the deos scheduler kernel. In *Proceedings of the 22nd International Conference on Software Engineering*. ACM Press, June 2000.
4. A. Cimatti, F. Giunchiglia, G. Mongardi, D. Romano, F. Torielli, and P. Traverso. Model Checking Safety Critical Software with SPIN: an Application to a Railway Interlocking System. Presented at SPIN97, the Third SPIN Workshop, April 1997 (online proceedings at `http://netlib.bell-labs.com/netlib/spin/ws97/papers.html`).
5. K. Havelund, M. Lowry, and J. Penix. Formal Analysis of a Space Craft Controller using SPIN. Presented at SPIN98, the 4th International SPIN Workshop, November 1998 (online proceedings at `http://netlib.bell-labs.com/netlib/spin/ws98/program.html`).
6. S. Vestal. Modeling and verification of real-time software using extended linear hybrid automata. To appear at Lfm2000, June 2000 (see `http://atb-www.larc.nasa.gov/fm/Lfm2000/`).