Model-Checking of Component-Based Event-Driven Real-Time Embedded Software [*]

Zonghua Gu

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Kang G. Shin

Department of EECS
University of Michigan
Ann Arbor, MI 48109

Abstract

As complexity of real-time embedded software grows, it is desirable to use formal verification techniques to achieve a high level of assurance. We discuss application of model-checking to verify system-level concurrency properties of component-based real-time embedded software based on CORBA Event Service, using Avionics Mission Computing software as an application example. We use the process algebra FSP to formalize specification of software components and system architecture, previously only available in the form of natural language and prone to misinterpretation and misunderstanding, and use model-checking to verify system-level concurrency properties. We also discuss effective techniques for coping with the state-space explosion problem by exploiting application domain semantics. We have applied our analysis techniques to realistic application scenarios provided by our industry partner to demonstrate their utility and power.

1. Introduction

The publish/subscribe model of computation, as implemented in CORBA Event Service [1], has been widely adopted in a variety of application domains, including both real-time embedded systems and enterprise distributed systems. One example is the *Avionics Mission Computing* (AMC) [2] software, which is the embedded software onboard a military aircraft for controlling mission-critical functions, such as navigation, target tracking and identification, weapon firing, etc. The software architecture of AMC is also commonly referred to as the *Bold Stroke Framework*. It is modeled with UML, but manually coded with C++. The UML models mainly serve in a documentation role that the engineer refers to while writing code manually. Therefore, the link between model and code is easily broken in the process of system maintenance and evolution, when code is modified or enhanced without making the corresponding changes of the model, or vice versa. Furthermore, UML has little support for analysis that is relevant for embedded systems, such as real-time and concurrency properties, such as schedulability and deadlock freedom. As part of the DARPA *Model-Based Integration of Embedded Software* (MoBIES) Program, an end-to-end tool-chain [3, 4, 5] has been developed collaboratively by researchers from Vanderbilt University, Southwest Research Institute, and University of Michigan. The MoBIES tool-chain covers the entire systems development life-cycle, including modeling, code generation and analysis, and provides a more automated and integrated development process than the current industry practice. The central repository of information in the tool-chain is the *Embedded Systems Modeling Language* (ESML) [6], a domain-specific language for modeling component-based, event-driven software using the *Generic Modeling Environment* (GME) [7] from Vanderbilt University. ESML is designed to be a comprehensive modeling language that captures essential aspects of embedded systems, including software architecture, timing and resource constraints, execution threads, execution platform (processors and network) information, allocation of components to threads/processors, etc. We have developed a tool AIRES [8] to perform various static analysis tasks on ESML models, such as dependency, timing, schedulability and automated component allocation.

ESML and AIRES mainly focus on the *static structural* aspects while largely ignoring the *dynamic behavioral* aspects of the embedded software. As a result, ESML models are not *executable*. In order to perform deeper semantic analysis, it would be necessary to construct *executable models*, which enables the use of simulation or model-checking to verify system correctness. One prominent example of executable models is Harel's Statechart [9]. Model-checking can be viewed as exhaustive simulation, i.e., exhaustively exploring the system state space to prove certain correctness properties.

Documentation provided by our industrial partner Boeing (not available to the public) describes the application components and scenarios with natural language, which is prone to misunderstanding and misinterpretation. In this paper, we use the process algebra *Finite State Processes* (FSP) [10] to provide formal specifications of dynamic behavioral aspects of the AMC software, and use the model-checker *Labeled Transition System Analyzer* (LTSA) [10] to analyze the resulting FSP specification and verify concurrency properties such as deadlock freedom, event reachability, sequencing constraints and progress property. We also exploit application domain semantics to cope with the state-space explosion problem. First, we reduce the call-return two-way synchronization into a one-way synchronization, thus reducing the number of states of each component. Second, we take advantage of inherent modularity within the application scenario, and use

∗ The work reported in this paper was supported in part by DARPA under contract F3615-00-1706, by ARO under grant DAAD19-1-1-0473, and by ESCHER.

the divide-and-conquer approach to compose the system hierarchically. These techniques showed significant benefits in reducing system state-space, and allowed us to check relatively complex application scenarios on a PC workstation with a relatively modest memory size of 512MB. When the system size is too large for the model-checker to handle, the designer can at least use simulation to gain some insight into system behavior.

Although our work is initially targeted towards the AMC software, it is applicable to more general component-based event-driven real-time embedded software. The AMC software architecture is very similar to the *CORBA Component Model* (CCM) [11], which was originally designed for enterprise applications, but has been recently extended to be real-time and QoS-enabled by researchers from Washington University to produce *Component-Integrated ACE ORB* (CIAO) [12]. In fact, there are plans to migrate the next-generation of AMC software to the CIAO platform. Vanderbilt University has developed a model-based toolset *Component Synthesis with Model Integrated Computing* (CoSMIC) [13] for design and configuration of applications based on the CIAO platform. Our approach could be easily adapted to apply to general CCM applications, for example, by generating FSP models from CoSMIC instead of ESML.

This paper is structured as follows. Section 2 provides a brief introduction to FSP. Section 3 provides a brief introduction to AMC. Section 4 describes modeling of AMC with FSP. Section 5 discusses the specification of correctness properties for verification. Section 6 presents techniques for improving model-checking scalability. Section 7 discusses related work, and Section 8 draws conclusions.

2. Introduction to Finite State Processes

We only provide a very brief description of FSP, and refer the interested reader to [10] for more details.

Primitive processes are defined as finite-state processes using event prefix `->`, choice | and recursion. If `x` is an event and `P` a process, then `(x->P)` describes a process that initially synchronizes with the event `x` and then behaves exactly as process `P`. If `x` and `y` are events, then `(x->P|y->Q)` describes a process which initially synchronizes with either `x` or `y`, and the subsequent behavior is described by `P` or `Q`, respectively. Primitive processes can be composed with the parallel composition operator || to form a *composite process*. Processes interact via synchronization on common event labels in the traditional style of process algebra. That is, if processes in a composition have a common shared event, all processes must synchronize on the shared event at the same step.

3. Introduction to Avionics Mission Computing

The AMC software consists of components interacting with each other using the publish/subscribe paradigm with Real-Time CORBA Event Service [1] as its underlying communications substrate. Event publisher components push events through the event channel to event consumer components, whose execution is triggered by the arrival of events. The system runs at a number of different rates driven by timer event publishers, such as 40Hz, 20Hz, 10Hz, 5Hz, and 1Hz. Thread priorities are assigned rate-monotonically, that is, higher frequency threads are assigned higher priorities. Rate Monotonic Analysis [14] is used to make real-time guarantees.

Components are composite objects with ports, interacting with one another either through event triggers or method invocations. Some terminologies from *CORBA Component Model* (CCM) are adopted. Each component can have the following types of ports:

- *Publish Port* to publish events.
- *Subscribe Port* to subscribe to events.
- *Receptacle* to issue method invocations.
- *Facet* to accept method invocations.

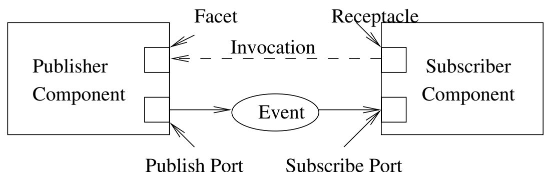

Figure 1. The control-push/data-pull style of interaction.

Component interaction typically (but not always) follows the *control-push/data-pull* style, as shown in Figure 1. The data producer component publishes a `DataAvailable` event from its publish port indicating that it has fresh data; when the data consumer component receives the event from its subscribe port, it issues a `GetData()` call from its receptacle to the producer's facet to retrieve the data.

4. Modeling AMC with FSP

4.1. Modeling of Component Types

The AMC software is component-based with many different types of components, each with its unique functionality and interfaces, acting as basic building blocks of a complete system. The documentation provided by our industrial partner contains detailed descriptions of the various component types in natural language. We use FSP to provide an unambiguous, formal description for each component *type* based on the natural language descriptions, and instantiate each component *instance* to form a system architecture. In what follows, we describe each component type by excerpting its description from the Boeing documentation, and then presenting its corresponding FSP specification. Note that this is not an exhaustive list of all component types, but only includes a few interesting ones from a modeling viewpoint.

- "*DisplayComponent* is used to display information to the console window. It is used to simulate any output device in a system. Upon receiving a Push(), this component does a Get() on each component specified in its receptacles. It then displays the results on the console."

```
DisplayComp =
(inEvt->issueGDCall->receiveGDReply->display
->DisplayComp).
```

- "*LazyActiveComponent* is used to simulate delayed response to acquiring data. As an optimization strategy, if a component is updated more often than it is read, the Lazy Active pattern may be used to only update the data when a request is made. Upon receiving a Push(), this component flags its data as invalid. When this component's Get() is called, this triggers the LazyActiveComponent to call Get() on the components attached to its receptacles."

```
LazyActiveComp = (inEvt->outEvt->DataStale
|receiveGDCall->issueGDReply->LazyActiveComp),

DataStale=
(receiveGDCall->issueGDCall->receiveGDReply
->issueGDReply->LazyActiveComp).
```

- "*ModalComponent* is used to alter the flow of events. The component can be enabled and disabled via the facet method ChangeMode(). When it is enabled, it will update and generate an event when it receives an event. When it is disabled, it will not update or generate an event."

```
ModalComp = Enabled,

Disabled = (enable->Enabled|disable->Disabled
|inEvt->Disabled),

Enabled = (enable->Enabled|disable->Disabled
|inEvt->issueGDCall->receiveGDReply->outEvt
->Enabled
|receiveGDCall->issueGDReply->Enabled).
```

4.2. Modeling of Component Interactions

4.2.1. Control-Push/Data-Pull Below is the FSP model for the control-push/data-pull interaction style as shown in Figure 1.

```
Publisher = (outEvt->Publisher |
receiveCall->issueReply->Publisher).

Subscriber = (inEvt->issueCall->receiveReply
->Subscriber).

||ControlPushDataPull =
(pub:Publisher||sub:Subscriber)
/{pub.outEvt/sub.inEvt,
sub.issueCall/pub.receiveCall,
sub.receiveReply/pub.issueReply).
```

This modeling approach treats the interaction between an event publisher and an event subscriber as *synchronous*, that is, the `outEvt` of the publisher synchronizes with the `inEvt` of the subscriber directly. In the real system, the published events go through the CORBA event service and are buffered at the input port of the subscribe component. We can obtain a more accurate model by using separate processes to model the queues/buffers in the middleware infrastructure, and decouple the interactions to make them asynchronous, but that will have a significant impact on scalability in terms of the maximum size of the system that the model-checker can handle. Note that our focus is on verification of *application-level* concurrency properties when the application is operating under normal conditions assuming that the middleware behaves correctly, with no buffer overflows or deadline misses (see Section 4.2.3 for more details on this assumption). This synchronous modeling style has turned out to be at an adequate level of abstraction for the types of concurrency properties we are interested in, e.g., deadlock freedom, event reachability, sequencing constraints, and progress property. That is, adopting a more detailed modeling approach would not change the verdict of the model-checker on these properties. This may not be generally true if we expand the range of properties to include other properties involving the middleware, for example, buffer overflow detection.

The words *synchronous* and *asynchronous* are overloaded terms with different meanings to different people. Here we use the word *synchronous* to mean that pairwise interactions, such as event delivery and method invocation, between components happen instantaneously without the delays introduced by the middleware infrastructure. This is very different from its meaning in *synchronous formalisms* such as Esterel, which describes a time-triggered system with a global clock tick, commonly found in hardware and safety critical software systems.

4.2.2. Input Event Correlation When a component subscribes to multiple events, there may be two synchronization patterns: `AND` synchronization means that the component must receive all input events to be triggered; `OR` synchronization means that the component only needs to receive one of the input events to be triggered. In order to model `AND` synchronization, we add a new process type called `InputANDCorrelator`, as shown below and in Figure 2:

```
Event(ID=1) = (inEvt[ID]->GotOne),
GotOne = (inEvt[ID]->GotOne |
matched->Event).

||InputANDCorrelator(NumInputs=2) =
(if(NumInputs>0) then
(forall [i:1..NumInputs] Event(i))).
```

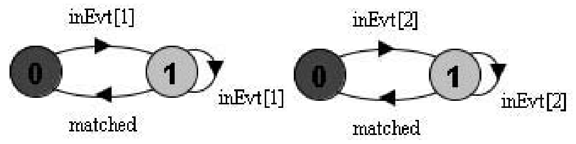

Figure 2. The correlator for two input events with AND correlation.

It models parallel composition of `NumInputs` number of processes `Event`, which all synchronize on the same event `matched`. This ensures that the `matched` event is emitted only when all input events `inEvt[i..NumInputs]` occur. The event `matched` event is in turn used to trigger the downstream subscriber component.

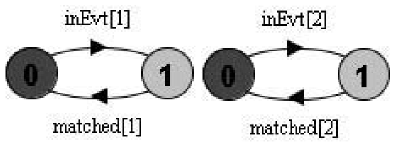

Figure 3. The correlator for two input events with OR correlation.

Input events may arrive at different rates. For example, a component may subscribe to `inEvt[1]` arriving at 20Hz rate, and `inEvt[2]` arriving at 1Hz rate. Only 1 out of every 20 `inEvt[1]` is paired up with 1 `inEvt[2]` to generate a `matched` event; the other 19 are silently discarded, as modeled by the self loop in state 1.

Modeling of `OR` input correlation is simpler, shown below and in Figure 3.

```
Event(ID=1) = (inEvt[ID]->matched[ID]->Event).

||InputORCorrelator(NumInputs=2) =
(if(NumInputs>0) then
(forall[i:1..NumInputs] Event(i))).
```

4.2.3. Real-Time Issues The typical way to model real-time in FSP is to discretize time into uniform segments by using a global event `tick` shared among all the processes in the system to provide a system-wide heartbeat. Typical component execution time in an AMC system is fairly small, in the range of microseconds, while the typical period of execution is fairly large, in the range of milliseconds or even seconds. If we attempt to model quantitative time by using a fine-grained partitioning of time on the microsecond scale, the system state space will quickly explode even for trivial examples. Instead, we only ensure that the relative execution frequencies of different rate groups are correct, e.g., the 20Hz thread should execute 20 times more frequently than the 1Hz thread. This can be achieved by using a shared event `tick`. If the event `timeout20hz` is emitted at every tick, then the event `timeout1hz` is emitted every 20 ticks.

```
Timer20hz = (timeout20hz->tick->Timer20hz).
Timer1hz = (timeout1hz->Delay20[1]),
Delay20[t:1..20] = (when(t==20)tick->Timer1hz
|when (t < 20) tick->Delay20[t+1]).
```

Note that the global event `tick` is only shared among all the timers, not the application components. Therefore, even though the periodic timer triggers synchronize to a system-wide heartbeat, application components interact with each other in an asynchronous, event-driven fashion.

Even though we cannot model quantitative time, we make the implicit assumption that the system is *schedulable*, i.e., all threads meet their deadlines. Without this assumption, we would have a much larger state space due to deadline misses, an error condition that should never arise in a production system, without gaining any additional insight into the system's normal operation. We can achieve separation of concerns by using AIRES to verify the schedulability assumption, and model-checking to verify concurrency properties. We encode this assumption in the model by adding an explicit synchronization between the timer and the *terminal events*, the leaf events of the event dependency graph rooted at the timeout event, in order to ensure that the next `timeout` event will not occur until all events belonging to the current execution frame have been processed. For example, timer-triggered sensor data may go through some processing stages and eventually trigger both the Flight Plan Display and the Navigation Display. We insert an AND correlator to make sure that both Display components have been triggered before the next 20Hz `timeout`.

```
Timer20hz = (timeout20hz->timer20hzDone->tick
->Timer20hz).

Thread20hz = (...
||fltPlanDisplay:displayComp
||navDisplay:displayComp
||correlator:InputANDCorrelator)
/{fltPlanDisplay.display/correlator.inEvt[1],
navDisplay.display/correlator.inEvt[2],
correlator.matched/timer20hzDone
}.
```

There may be *aperiodic* and *sporadic* external interrupts that act as thread triggers in addition to periodic timers. For sporadic interrupts, there is a bound of *minimum inter-arrival time* (MIAT) between interrupts, so we make a pessimistic assumption and model the interrupt source as a periodic timer with period equal to MIAT, as is commonly done in real-time scheduling analysis. For aperiodic interrupts, we make the interrupt source *not* synchronize with the global event `tick`, meaning that the interrupts can happen at arbitrary points in time without any timing constraints, which is exactly the definition of aperiodic interrupts.

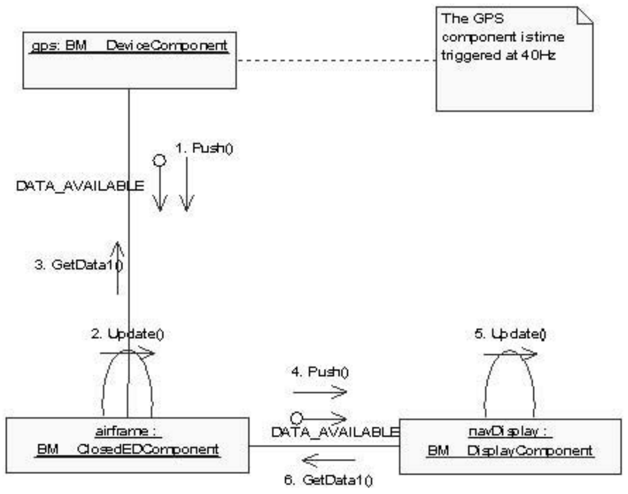

Figure 4. The Basic Single-Processor (BasicSP) scenario.

4.2.4. An Example Application Scenario As an illustrative example, we consider the *Basic Single-Processor* (BasicSP) application scenario in Figure 4. At a 40Hz rate, the system must update navigation displays with timely airframe position information using inputs from navigation sensors. Triggered by the 40Hz timer, the gps component pushes a DataAvailable event to the airframe component, which updates its state by getting data from gps. The airframe component then pushes a DataAvailable event to the navDisplay component, which then updates the display by getting data from airframe. (The prefix BM for the component types is a naming convention, meaning that these component types are basic *Building Block* models for any application, as opposed more application specific components such as OM, for *Operator Interface* models. We omit these prefixes in FSP specifications.) Below is the complete FSP specification for this scenario:

```
Timer40hz = (timeout40hz->timer40hzDone->tick
->Timer40hz).

DeviceComp = (inEvt->outEvt->DeviceComp |
receiveGDCall->issueGDReply->DeviceComp).

ClosedEDComp =
(inEvt->issueGDCall->receiveGDReply->outEvt
->ClosedEDComp
|receiveGDCall->issueGDReply->ClosedEDComp).

DisplayComp =
(inEvt->issueGDCall->receiveGDReply->display
->DisplayComp).

||Thread40hz = (Timer40hz
||gps:DeviceComp
||airframe:ClosedEDComp
||navDisplay:DisplayComp)
/{timeout40hz/gps.inEvt,
gps.outEvt/airframe.inEvt,
airframe.issueGDCall/gps.receiveGDCall,
airframe.receiveGDReply/gps.issueGDReply,
airframe.outEvt/navDisplay.inEvt,
navDisplay.issueGDCall/airframe.receiveGDCall,
navDisplay.receiveGDReply/airframe.issueGDReply,
navDisplay.display/timer40hzDone }.
```

LTSA has built-in functionality to perform simulation as user-controlled animation. Once the models are developed, we can use interactive simulation to gain deeper understanding of the system dynamics, or model-checking to verify concurrency properties. We will not elaborate on simulation due to space limitations. Instead, we will focus on model-checking in the following sections.

5. Specification of System Properties

Generally, model-checking can be used to verify two types of properties: *safety* and *liveness*. A safety property asserts that nothing bad happens, and a liveness property asserts that something good eventually happens. We consider the following safety properties: *deadlock freedom*, *event reachability* and *sequencing constraints*. We consider one liveness property related to *progress*. Besides these generic properties, it is possible to specify and verify other application-specific properties, for example, a certain component method is only invoked after a number of other component methods are invoked for a specific number of times and in a specific order.

5.1. Deadlock Freedom

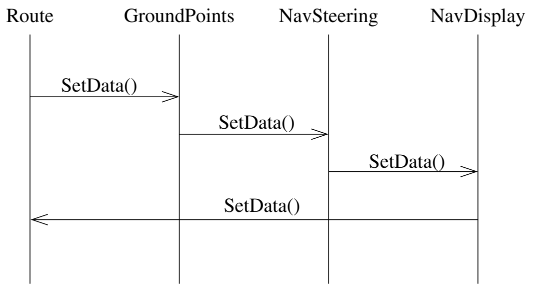

Figure 5. A deadlock situation caused by a dependency cycle.

The most important safety property is deadlock freedom. Let's take an scenario of four components forming a chain of *data-push* interactions, as shown in Figure 5. For illustration purposes, we artificially introduce a deadlock situation by adding an extra method call from navDisplay to route, as shown in Figure 5. When the route component's SetData() call is invoked, it is still blocked waiting for its method invocation to groundPoints to return. This is the classic deadlock situation caused by a circular dependency. LTSA detects this error and produces an error trace that leads to the deadlock.

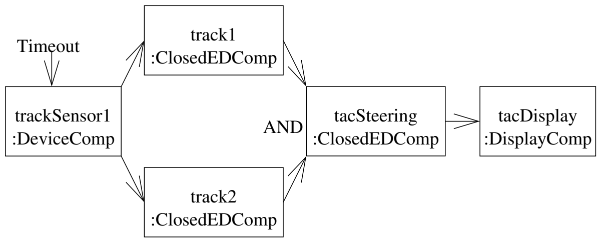

Figure 6. An application scenario fragment.

Another error situation arises when a component subscribes to multiple input events with AND synchronization, i.e., all input events must be received in order to trigger the component execution, but for certain reasons not all of the input events are available. Event propagation stops at this component, and none of the downstream components can be triggered. This situation is not a deadlock in the classic computer science sense, but it causes a deadlock in the FSP model due to our modeling approaching of synchronization of the leaf component actions with the timerdone event, discussed in Section 4.2.3.

Figure 6 shows an example application scenario. Component trackSensor is triggered periodically by the 20Hz timeout, and issues an output event that is subscribed to by both `track1` and `track2`. Component `tacSteering` subscribes to output events of both `track1` and `track2` with AND synchronization, and issues an event to trigger `tacDisplay`. LTSA reveals no deadlocks in this scenario.

If we make a change to the system by letting `trackSensor1` publish two types of events `outEvt1` and `outEvt2`, choosing non-deterministically which event to output at runtime. This could be used to model a modal component which outputs different events depending on its active mode. This change results in a deadlock situation detected by LTSA, since `tacSteering` can only receive one of its two input events during each execution cycle.

5.2. Event Reachability

Each component should be triggered/invoked at least once during each execution cycle. Otherwise, the component is redundant, which could signal a design error or inefficiency that wastes system resources. In order to prove that a component's method `C.m` is reachable, we introduce a property `NotReachable` stating that the event `C.m` never occurs. If this property holds, then `C.m` is indeed not reachable; otherwise, LTSA returns a counter example showing the path of execution leading to the event `C.m`. For example, if we would like to check that the action `navDisplay.display` is executed/reachable, we add the following:

```
property NotReachable = STOP+{reachable}.

||CheckReachability = (System || NotReachable)
/{navDisplay.display/reachable}.
```

Checking this property for the MultirateSP scenario yields this chain of events that lead to the triggering of `navDisplay.display`:

```
Trace to property violation in NotReachable:
    timeout40hz
    gps.outEvt
    airFrame.issueGDCall
    airFrame.receiveGDReply
    airFrame.outEvt
    navDisplay.issueGDCall
    navDisplay.receiveGDReply
    navDisplay.display
```

This means that `navDisplay.display` is indeed reachable, which is the correct behavior.

5.3. Sequencing Constraints

Certain events should happen in sequence. For example, the events in a linear chain of event triggers should happen in the order of precedence relation from the head to the tail of the chain. Below is the property specification used to check the correct ordering of events in the 40Hz thread:

```
property SeqConstraint =
(evt1->evt2->evt3->evt4->SeqConstraint).
||CheckSeqConstraint = (SYSTEM||SeqConstraint)
/{timeout40hz/evt1, gps.outEvt/evt2,
airframe.outEvt/evt3, navDisplay.display/evt4
}.
```

LTSA reports no violations for this property. Suppose we change the sequencing order of `gps.outEvt` and `airframe.outEvt`:

```
property SeqConstraint =
(evt1->evt2->evt3->evt4->SeqConstraint).
||CheckSeqConstraint = (SYSTEM||SeqConstraint)
/{timeout40hz/evt1, airframe.outEvt/evt2,
gps.outEvt/evt3, navDisplay.display/evt4}.
```

Then, LTSA produces an error trace:

```
Trace to property violation in SeqConstraint:
    timeout40hz
    gps.outEvt
```

If some events may happen in parallel, that is, the events form a general graph instead of a linear chain, then we can only specify those events that do form a linear chain, since LTSA does not allow non-determinism in property specifications.

5.4. Progress Property

The properties discussed so far are all *safety properties*, that is, they can be verified by detecting if a bad state is reached given a *finite* execution sequence. On the other hand, *liveness properties* can only be verified for an infinite execution sequence. A general treatment of liveness involves using temporal logic to specify liveness properties. A restricted class of liveness properties is the *progress* property in the form of `progress P = a1, a2, ..., an`, which asserts that in an infinite execution of a system, at least one of the actions `a1, a2, ..., an` will be executed infinitely often. It is useful for verifying that a system does not contain starvation of certain actions. It is a stronger assertion than reachability, which only requires that certain actions are executed *at least once* during the system's lifetime.

For example, in order to check that the `display` methods of both Flight Plan Display and Navigation Display are executed infinitely often in any infinite execution of the MultirateSP scenario, we can add this to its FSP model:

```
progress P1 = {fltPlanDisplay.display}
progress P2 = {navDisplay.display}
```

Note that this is different from:

```
progress P1 = {fltPlanDisplay.display,
navDisplay.display}
```

which states that at least one of `fltPlanDisplay.display` and `navDisplay.display` are executed infinitely often.

6. Scalability Improvements

The biggest impediment to industry adoption of model-checking is lack of scalability due to state-space explosion. We have constructed the FSP model for the *Medium Single-Processor* (MediumSP) scenario [8], which consists of two threads running at 20Hz and 1Hz. This scenario causes *out-of-memory* error on a state-of-the-art PC workstation. We discuss techniques for improving scalability of model-checking by exploiting application domain semantics. That is, certain characteristics of the application domain allow us to perform optimizations and reduction of the model that is input to the model-checker. After making these improvements, we were able to compose and check this application scenario.

6.1. Atomic Method Call/Return

Normally method calls are modeled with a two-way synchronization between the *caller* component and the *callee* component. However, for all practical purposes we can treat the `GetData()` call and reply as an atomic operation, and omit the synchronization action on the method call reply. This optimization may not be generally applicable to all method calls, but only to the `GetData()` call in the *control-push/data-pull* interaction style, where there is no action in between the GetData() call and reply. This involves modifying definition of each component type. After applying this optimization, the state space of the MediumSP scenario has been reduced considerably. However, this is still too large for LTSA to handle on our PC workstation.

6.2. Compositional Analysis

Construction of the global state space of an application usually causes state-space explosion. AMC application scenarios often exhibit modularity, that is, certain groups of components have intensive communication and interaction among themselves, but relatively little interaction with components outside the group. We can take advantage of inherent modularity within the application to compose and check the system hierarchically, instead of composing the entire system state-space all at once. After a set of components have been checked to be correct, we can abstract and reuse them in other contexts by hiding irrelevant events and only exposing those events that may be of interest to other surrounding components, resulting in a simplified and minimized automaton. We can then reuse this automaton as a module in other contexts. This is the typical divide-and-conquer approach. The compositional analysis technique allowed us to compose and check the MediumSP scenario successfully.

However, there is one drawback of the compositional analysis approach. Since internal events are hidden inside of each group of components, we cannot check for end-to-end sequencing constraints that span multiple groups and involves internal events from these groups. We can only check constraints that involve interface events that are exposed by the component group, or those that involve internal events of a single group. All the other properties are not impacted.

6.3. Performance Evaluation

We have applied model-checking to a number of application scenarios. The experiments were performed on a PC workstation with 512MB of memory and Pentium IV processor running Windows XP. Obviously, using a more powerful computer with more memory would help improve scalability. The scenarios range from the BasicSP scenario with 3 components, to the MediumSP scenario with more than 50 components. However, scenarios larger than MediumSP are still beyond the reach of the model-checker despite our state-space reduction techniques. Model-checking generally finishes within seconds or at most a few minutes when the main memory is large enough; otherwise, the computer goes into virtual memory thrashing mode and eventually gives out the *memory exhausted* error. We believe 50+ components is a reasonable size to make this approach useful. A realistic AMC system has up to thousands of components, and at present, no model-checker can be expected to be able to scale up to that size. We will have to rely on the designer's manual work to separate out fragments of scenarios that are relatively isolated from the rest of the system and model-check them individually. This is a very reasonable thing to do, as the current application scenarios are just fragments taken from a production system.

Not surprisingly, we have found no errors in these application scenarios, which are fragments taken from a mature, tried-and-true production system. However, we believe the model-checking approach can act as a valuable debugging tool for uncovering subtle concurrency bugs during the early design stage of a new system, or the maintenance stage of a legacy system.

7. Related Work

The model-checker Bogor [15] is an extensible model-checker with built-in support for OO structures and communication layers, integrated with Cadena [16] for verification of functional properties, also targeting the Avionics Mission Computing software. Garlan [17] described a model-checking framework for publish/subscribe systems. The key feature of this framework is a reusable, parameterized state machine model that captures pub-sub runtime event management and dispatch policy. Generation of models for specific systems is then handled by a translation tool that accepts as input a set of component descriptions together with a set of properties, and maps them into the input format of the model-checker SMV [18]. Compared to [15] and [17], our model-checking approach adopts a higher level of abstraction and ignores details related to the internals of middleware such as queuing and dispatch policies. This significantly reduces system state-space, and has turned out to be adequate for our purpose of verifying application-level concurrency properties, assuming that the middleware behaves correctly. We also take advantage of application domain semantics and LTSA's compositional analysis capability to help improve scalability, which is not present in [15] and [17]. Some of the property specifications, such as sequencing constraints and progress property, are also unique to our approach.

Madl [19] developed automated translation from ESML to Timed Automata, and used the model-checker UPPAAL [20] to verify real-time properties. One limitation of their approach is that they can only model *non-preemptive* scheduling within a single rate group/thread, but not *preemptive* scheduling between threads, due to lack of expressive power (a stopwatch mechanism) of the Timed Automata formalism. The preemption effects of higher-priority threads obviously have an impact on the timing behavior of the lower-priority threads since they share the same CPU. As discussed in Section 4.2.3, we believe a more practical approach is to achieve separation of concerns by using real-time scheduling theory to verify the schedulability assumption, and model-checking to verify concurrency properties.

Karamanolis [21] used FSP to model and verify workflow schemas by mapping workflow schemas into FSP models. There are some similarities between the computational models of workflow schemas and AMC software. Both consist of components interacting with events sent and received from output and input ports. However, there are also important differences due to the different application domains. For example, AMC software typically contains several threads executing periodically, while the life-cycle of a workflow schema only consists of one execution from start to finish. The different execution frequencies of multiple threads cause the state space of an AMC application to be much larger than a workflow schema specification with similar complexity. From a modeling perspective, a self-loop has to be added to each legal terminating state in [21] in order to avoid false alarms when checking for deadlocks, while this is not necessary for AMC since it is a reactive system that should never terminate.

8. Conclusions

In this paper, we have discussed application of model-checking to verify system-level concurrency properties of component-based real-time embedded software based on CORBA Event Service, with Avionics Mission Computing as one application example. Using the process algebra FSP, we were able to formalize specification of software components and system architecture, and use model-checking to verify concurrency properties such as deadlock freedom, event reachability, sequencing constraints, progress property and application-specific properties. We also discussed effective techniques for coping with the state-space explosion problem by exploiting application domain semantics. We have applied our analysis techniques to realistic application scenarios provided by our industry partner to demonstrate their utility and power. The publish/subscribe architectural style of AMC is widely-adopted for large scale embedded software, due to its nice property of decoupling between publishers and subscribers. Although our work is initially targeted towards the AMC software, it has much wider applicability to the general class of component-based event-driven real-time embedded software. As part of our future work, we plan to adapt our technique to be applicable to general applications based on the CORBA Component Model [11, 12].

References

[1] D. Schmidt, D. Levine, and T. Harrison, "The design and performance of a real-time CORBA object event service," in *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1997, pp. 434–445.

[2] D. Sharp, "Object-oriented real-time computing for reusable avionics software," in *Proc. IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2001, pp. 185–192.

[3] Z. Gu, S. Wang, S. Kodase, and K. G. Shin, "Multi-view modeling and analysis of embedded real-time software with meta-modeling and model-transformation," in *Proc. IEEE International Symposium on High Assurance Systems Engineering*, 2004, pp. 32–41.

[4] ——, "An end-to-end tool chain for multi-view modeling and analysis of avionics mission computing software," in *Proc. IEEE Real-Time Systems Symposium (RTSS)*, 2003, pp. 78–81.

[5] Z. Gu and Z. He, "Real-time scheduling techniques for implementation synthesis from component-based software models," in *Proc. ACM SIGSOFT International Symposium on Component-Based Software Engineering (CBSE)*, 2005.

[6] G. Karsai, S. Neema, A. Bakay, A. Ledeczi, F. Shi, and A. Gokhale, "A model-based front-end to tao/ace," in *Proc. Second Workshop on the ACE ORB (TAO)*, 2002. [Online]. Available: http://www.cs.wustl.edu/ schmidt/TAOWS02/

[7] A. Ledeczi, M. Maroti, G. Karsai, and G. Nordstrom, "Metaprogrammable toolkit for model-integrated computing," in *Proc. IEEE International Conference on Engineering of Computer-Based Systems*, March 1999, pp. 311–317.

[8] Z. Gu, S. Kodase, S. Wang, and K. G. Shin, "A model-based approach to system-level dependency and real-time analysis of embedded software," in *Proc. IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2003, pp. 78–85.

[9] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, June 1987. [Online]. Available: citeseer.nj.nec.com/harel87statecharts.html

[10] J. Magee and J. Kramer, *Concurrency: State Models and Java Programs*. Wiley, 2000.

[11] OMG, "CORBA Component Model, v3.0," Object Management Group, Tech. Rep., 2002.

[12] N. Wang and C. Gill, "Improving real-time system configuration via a qos-aware corba component model," in *Proc. IEEE Hawaii International Conference on Systems Sciences*, 2004, pp. 273–282.

[13] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt, "A platform-independent component modeling language for distributed real-time and embedded systems," in *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium*, 2004.

[14] C. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[15] W. Deng, M. Dwyer, J. Hatcliff, G. Jung, Robby, and G. Singh, "Model-checking middleware-based event-driven real-time embedded software," Kansas State University, Tech. Rep. SAnToS-TR2003-2, April 2003.

[16] J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad, "Cadena: An integrated development, analysis, and verification environment for component-based systems," in *Proc. IEEE International Conference on Software Engineering*, 2003.

[17] D. Garlan, S. Khersonsky, and J. S. Kim, "Model checking publish-subscribe systems," in *Proc. International SPIN Workshop on Model Checking of Software*, 2003, pp. 166–180.

[18] K. McMillan, *Symbolic Model Checking*. Kluwer Academic Publishing, 1993.

[19] G. Madl, S. Abdelwahed, and G. Karsai, "Automatic verification of component-based real-time corba applications," in *Proc. IEEE International Real-Time Systems Symposium*, 2004.

[20] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL - a tool suite for automatic verification of real-time systems," in *Proc. Workshop on Verification and Control of Hybrid Systems*, October 1995, pp. 232–243.

[21] C. Karamanolis, D. Giannakopoulou, J. Magee, and S. M. Wheater, "Model checking of workflow schemas," in *Proc. IEEE International Enterprise Distributed Object Computing Conference*, 2000, pp. 170–179.