

Model-checking Middleware-based Event-driven Real-time Embedded Software*

William Deng, Matthew B. Dwyer, John Hatcliff, Georg Jung, Robby, and
Gurdip Singh

Department of Computing and Information Sciences, Kansas State University **

Abstract. Component frameworks such as the CORBA Component Model (CCM) and middleware services such as the CORBA Event Service are increasingly being used to build safety/mission-critical distributed real-time embedded (DRE) systems. In this paper, we present a novel model-checking infrastructure for checking global temporal properties of DRE systems built on top of a Real-Time CORBA Event Service using CCM architectures. We describe how (a) building support for OO structures and communication layers directly in an extensible model-checker and (b) leveraging domain properties related to priorities, scheduling, and timing can dramatically reduce the costs of checking realistic systems.

1 Introduction

Modern distributed systems are often built using sophisticated component and middleware frameworks such as Enterprise Java Beans, the CORBA Component Model (CCM), and Microsoft's .NET. Moreover, real-time versions of these frameworks such as RT-CORBA and CCM increasingly are being used to build safety/mission-critical Distributed Real-time Embedded (DRE) systems [9].

Figure 1 displays the typical architecture of these systems: loosely-coupled components communicate through middleware layers that hide the complexities associated with moving data across network connections. The implementations of both the components and the middleware itself make extensive use of object-oriented (OO) features and design patterns to facilitate reuse. Middleware frameworks include sophisticated *services* to support functions commonly required in distributed systems such as transactions, persistence, etc. In particular, loose coupling of components is often achieved using asynchronous/event-based communication infrastructures such as the CORBA Event Service. Event services allow components to easily plug and unplug into the system via publish/subscribe mechanisms, and they provide support for defining event types, event filtering, and event correlation. Real-time event services [12] also provide capabilities for specifying real-time and quality of service attributes. Moreover, unlike typical concurrent architectures where each component might include one or

* This work was supported in part by the U.S. Army Research Office (DAAD190110564), by DARPA/IXO's PCES program (AFRL Contract F33615-00-C-3044), by Rockwell-Collins, by Intel Corporation (Grant 11462), and by Honeywell Technology Center and NASA Langley Research Center (NCC-1-399).

** 234 Nichols Hall, Manhattan KS, 66506, USA.

{deng,dwyer,hatcliff,jung,robby,singh}@cis.ksu.edu
Technical Report SAnToS-TR2003-2

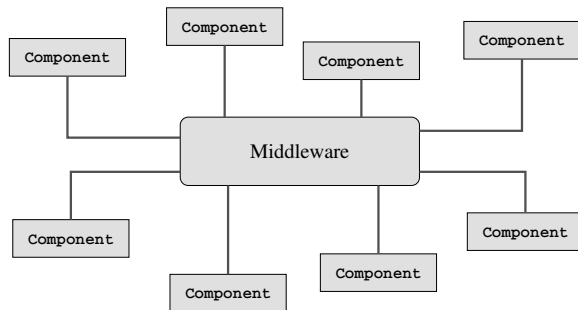


Fig. 1. The middleware architecture of a distributed system

more threads, real-time event services often achieve greater control of scheduling and timing constraints by providing a *thread pool*. In this approach, components are *passive* (they include no threads). Instead, the event service itself provides and manages threads which execute event-handlers of subscribing components when an event is published.

We are interested in reasoning about component-based DRE systems using model-checking techniques. Previous work that is relevant for this task falls roughly into two categories: (1) high-level modeling and analysis of asynchronous systems, and (2) recent work on checking OO software. First, high-level modeling and verification of asynchronous systems as embodied by tools such as Spin [15] has been a prominent theme of research in computer-aided verification. However, most existing tools for modeling distributed systems lack the direct support for OO features such as dynamically created objects, component connections, etc. that are ubiquitous in modern distributed systems. Moreover, the state representation and state exploration techniques in existing tools are general-purpose, and thus are not optimized to take advantage of numerous opportunities for reduction that could be enabled by considering, e.g., particular threading models or scheduling policies of DRE systems. Second, recent research has produced checking tools such as Bandera [5], Java Path Finder (JPF) [3], and dSpin [7] that directly support checking of concurrent systems implemented in OO languages by providing built-in representations of references, dynamically created objects, garbage collection, etc. However, despite the popularity of component-based frameworks and their potential to be utilized in mission- and safety-critical applications, relatively little has been done to enhance existing software model-checking techniques to provide native support for the complex event-based middleware of distributed component systems, as well as to exploit the specific properties of DRE systems to scale up analysis to feasibly approach real life problems.

To address this lack of support, we have built *Cadena* [13] – a development and verification environment for building DRE systems using CCM. Cadena supports specification of components using the CCM Interface Definition Language (IDL) with extensions to enable multiple forms of light-weight specification of component behavior and dependencies. Cadena, with its sophisticated GUI for component configuration and selection of real-time middleware configuration parameters, design-level slicing and dependence checks, along with extensive auto-coding facilities all presented inside IBM’s Eclipse open-source develop-

ment environment [10] provides support for end-to-end development of real-time CCM systems. In addition to the light-weight static analyses mentioned above, we have also built a model checking backbone into Cadena, focusing on safety and event-sequencing properties.

In our previous work [13], Cadena translated CCM system descriptions to the input language of the dSpin model-checker. dSpin directly supports references, and dynamic creation and deletion of objects, and therefore is well-suited for modeling the OO features of modern distributed systems. However, dSpin does not provide direct support for the threading model and scheduling policies of real-time middleware, nor does it have state-space reduction techniques that are tailored for these systems. In this paper, we describe a new model-checking core for Cadena that (a) provides several substantial advances in the representation of middleware models over our previous work and (b) significantly increases our ability to model and check properties of CCM-based DRE systems. Specifically, the contributions of this paper are as follows.

- We introduce a new model-checker called *Bogor* that, in addition to supporting direct modeling of OO software with a variety of sophisticated state space reduction techniques, also includes a powerful extension mechanism that allows new primitives to be added to the modeling language.¹
- We describe how to model a Real-Time CORBA Event Service using Bogor’s extension facilities. Compared to previous approaches for modeling publish/-subscribe systems [11], our approach allows much more direct modeling of realistic systems.
- We describe how (Bold Stroke) CCM architecture descriptions with light-weight behavioral annotations can be realized as Bogor models.
- Further, we present a series of strategies for leveraging timing/scheduling properties of soft real-time systems via Bogor primitives that allow checking of many systems that would otherwise be infeasible to check.

This checking infrastructure is being used to check avionics systems from Boeing and Rockwell-Collins. In this paper, we focus on Boeing’s Bold Stroke application framework [9] and discuss how our strategy is influenced by the actual Bold Stroke development process. In fact, we believe that this “customer-driven” context is one of the things that makes this work interesting and relevant: we address analysis of widely-used general purpose middleware frameworks and languages, and we design the functionality and features of our analysis tools to mesh with an actual industrial development process. This close cooperation with industry gives us better feedback about the feasibility of the introduced techniques than any purely theoretical approach could provide. We further gain insight in how to exploit domain information and scheduling policies to scale up model checking significantly. The methods introduced here suggest general approaches for exploiting domain info that may be adaptable to other component architecture models. A detailed formalization of the model-checking strategy that employ is presented in the extended version of this paper [8]. Here, due to space constraints, we focus on giving motivation and an overview of the implementation of our framework.

¹ We have completed a robust implementation of Bogor [22].

Section 2 gives a brief overview of Bold Stroke, describes which aspects of the Bold Stroke development process that we attempt to support, and presents our strategy for modeling Bold Stroke system behavior. Section 3 gives an overview of CCM and Cadena's support for development of DRE applications using CCM. Section 4 describes the CORBA Real-Time Event Channel and highlights factors that influence the design of appropriate models. Section 5 presents our strategy for modeling Bold Stroke systems using Bogor. Section 6 reports on experimental studies that we have carried out to validate our approach. Section 7 discusses related work, and Section 8 concludes.

2 Bold Stroke and Our Modeling Approach

2.1 Bold Stroke

Boeing's Bold Stroke program is an example where CORBA middleware has been embraced in a DRE domain for the reasons outlined above [9]. Bold Stroke is a product-line based program providing object-oriented mission critical avionics software to a variety of military aircraft produced by the Boeing company. Avionics software acts as the center of mission control for an aircraft pilot. It manages the cockpit displays, navigation and tactical sensors as well as weapon deployments. These complex systems have hard and soft real-time deadlines involving large amounts of periodic and aperiodic processing, and support thousands of operating modes. In addition, the software developed for military aircraft is maintained and updated over the course of many years. Although the development process is repeated for each update, each update aims to preserve as much legacy software as possible to reduce cost and risk. Bold Stroke represents a significant technological advance over Boeing's previous mission computing development practices which were largely assembly code based.

There are many aspects of a Bold Stroke system's functional and real-time behavior that we do not attempt to model in Cadena. We choose to focus on supporting the *system design and assembly phase* which Boeing engineers have pin-pointed as being one of the most challenging aspects of system construction. In this phase, a *component integrator* attempts to satisfy the functional and soft real-time requirements of a system by (a) hooking together general-purpose and project-specific components drawn from a component library and (b) selecting distribution strategies, execution priorities, and particular event/data communication layers. Rate-monotonic scheduling theory and conventional schedulability analysis techniques are employed to ensure that real-time deadlines are achieved. However, the inability to reason about high-level control-flow and abstractions of the system data state often leads to costly iterations in the design-code-test process. Specifically, engineers desire support for reasoning about (a) intra-component control-flow realized by conventional control-constructs (conditionals, locking, etc.), (b) the mode states of components and the effect that these mode states have on enabling/disabling particular component actions or communication patterns, and (c) inter-component control-flow realized by event subscription patterns and orderings of broadcast, reception, and correlation of events, as well as method calls.

2.2 Modeling approach

Building the approach of Garlan and Khersonsky [11] for model checking publish-subscribe systems, we factor Cadena system descriptions into three parts:

- (1) a collection of semantic descriptions for the components that make up the system (these are developer-specified, application dependent and capture aspects (a) and (b) above), and
- (2) a collection of reusable models of run-time event-delivery infrastructure (these are provided to the developer, they are re-used in each application that Cadena supports, and they capture the semantics of inter-component communication identified in aspect (c) above), and
- (3) a collection of connection actions that specifies the connection topology of the components and hooks the component models of part (1) to the middleware models of part (2).

Component models: Component models are defined using a simple transition-based modeling language that is similar to, e.g., Promela [15] but also includes object references and method calls. Modeling intra-component control-flow as required by aspect (a) above is straightforward using the control constructs of this language. Reasoning about system mode states as required by aspect (b) fits nicely with verification by model-checking since mode variables have small finite domains (e.g., a component is an `enabled` or `disabled` mode). We model such modes using enumerated types in our modeling language. In summary, our component models need only include simple control-flow skeletons with transition actions consisting of reads/writes of mode variables, event publish/receives, and method calls to local objects.

Middleware infrastructure models: Modeling inter-component communication is more difficult since the semantics of CORBA communication layers must be captured at a level of abstraction that is fine enough to expose interleavings that can lead to property violations, but also coarse enough to avoid state-space explosion for systems with a large number of components. We define several variants that trade precision for space/time to varying degrees using Bogor's extension facilities. This involves defining Bogor extensions to represent priority-based event queues and customizing Bogor modules to the particular scheduling strategies used in the RT CORBA middleware. When forming a system model, the developer chooses a particular variant for the middleware model from a library provided by Cadena.

Connection actions: Bogor's native support for method calls, dynamic object creation, and object references allows components to be connected to the communication layer by passing object references in a manner that closely follows the actual implementation. Accordingly, system initialization is modeled by a sequence of Bogor object-creation statements to create models components and middleware services, followed by a sequence of connection actions that pass appropriate references to establish connectivity.

```

#pragma prefix "cadena"
module modalsp {
  interface ReadData {
    readonly attribute any data;
  };

  eventtype TimeOut {};
  eventtype DataAvailable {};

  enum LazyActiveMode {stale , fresh};
  component LazyActive {
    provides ReadData dataOut;
    uses ReadData dataIn;
    publishes DataAvailable
      outDataAvailable;
    consumes DataAvailable
      inDataAvailable;
    attribute LazyActiveMode dataStatus;
  };

  enum OnOffMode {enabled , disabled};
  interface ChangeMode {
    attribute OnOffMode modeVar;
  };

  component Modal1 {
    provides ChangeMode modeChange;
    provides ReadData dataOut;
    uses ReadData dataIn;
    publishes DataAvailable
      outDataAvailable;
    consumes DataAvailable
      inDataAvailable;
  };
};

```

Fig. 2. CCM/Cadena artifacts for ModalSP (excerpts)

3 CORBA Component Model and Cadena

3.1 CCM Component Interface Definitions

In the CCM architecture, a system is realized as a collection of components. Each component has a *component interface* consisting of one or more *ports* that are used to connect to other components. There are two different kinds of connections – *interface* connections and *event* connections. Each port is unidirectional, so there are four types of ports: an interface supplier (a *facet* port), an interface consumer (a *receptacle* port), an event supplier (an *event source* port), and an event consumer (an *event sink* port). The CCM *interface definition language* (IDL) is used to define component interfaces consisting of named ports as well as interface and event types used as port types.

Figure 2 gives the CCM IDL that defines the interfaces for two component types called `LazyActive` and `Modal1` (these component types are used in an example of a simple avionics system called ModalSP that we define below). We use these definitions to illustrate the basic mechanisms for interface and event connections.

The type of an interface connection is defined by an interface definition – a collection of method signatures corresponding to the conventional notion of *interface* in Java or CORBA. Interfaces provide a mechanism for components to exchange data via synchronous method calls. Frequently, an interface will contain an *accessor* method `get_F` and a *mutator* method `set_F` to manipulate data associated with a particular data field *F*. Such methods are abbreviated by declaring field *F* to be an *attribute* of a particular interface (the IDL compiler will then automatically generate the accessor/mutator methods). For example, the `ReadData` interface of Figure 2 contains a single attribute `data` (the `readonly` qualifier indicates that only the accessor method `get_data` should be in the interface).

The type of an event connection is defined by an event type declaration – essentially, a record or structure containing zero or more data fields. For example, the `TimeOut` and `DataAvailable` are event types that both have zero data fields (in our application, we will only be interested in the arrival of such events – no payload is needed).

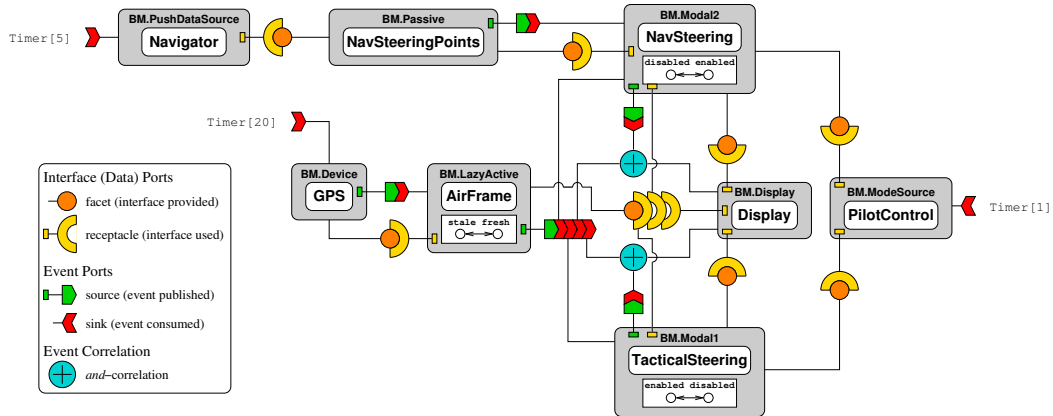


Fig. 3. Simple avionics system

The `LazyActive` component type declares four ports: it *provides* a facet interface of type `ReadData` with the name `dataOut`, it *uses* a receptacle interface of type `ReadData` with the name `dataIn`, it *publishes* an event of type `DataAvailable` through the port `outDataAvailable` and it *consumes* an event of type `DataAvailable` on port `inDataAvailable`. Additionally, the `LazyActive` component declares a component mode variable via an *attribute* `dataStatus` of type `LazyActiveMode`. In general, attributes are used either in component configuration or to represent some other aspect of component state.

3.2 CCM Systems

Figure 3 presents the CCM architecture for a very simple avionics system that shows steering cues on a pilot’s navigational display. The pilot can choose between two different display modes: a *tactical* display mode displays steering cues related to a tactical (*i.e.*, mission) objective, while a *navigation* display mode displays cues related to a navigational objective. Cues for the navigation display are derived in part from navigation steering points data that can be entered by the navigator.

The system is realized as a collection of components coupled together via interface and event connections. Input position data is gathered periodically at a rate of 20 Hz in the `GPS` component and then passed to an intermediate `AirFrame` component (which in a more realistic system would take position data from a variety of other sensors). Both the `NavSteering` and `TacticalSteering` component produce cue data for `Display` based on air frame position data. The `Navigator` component polls for inputs from the plane navigator at a rate of 5 Hz that are used to form `NavSteeringPoints` data. This data is then used to form navigational steering cues in `NavSteering`. `PilotControl` polls for a pilot steering mode at a rate of 1 Hz and enables or disables `NavSteering` and `TacticalSteering` accordingly. The time between each occurrence of a timeout of rate r is referred to as the *frame* of r , e.g., the length of the frame associated with the 5 Hz rate is 200 milliseconds.

There are several architectural aspects of this example that are peculiar to real-time and Bold-Stroke applications. First, note that periodic processing is achieved by having a component such as `GPS` subscribe to a periodic time-out

```

component LazyActive {
  mode status represents
    LazyActive.dataStatus init stale;

  behavior {
    any dat;
    push_inDataAvailable(DataAvailable e) {
      state := stale;
      outDataAvailable(e);
    }
    any dataOut.get_data() {
      if(status == stale) {
        dat <- dataIn.get_data();
        state := fresh;
      }
      return dat;
    }
  }
}

component Modal1 {
  mode onOff of OnOffMode init enabled;

  behavior {
    any dat;
    push_inDataAvailable(DataAvailable e) {
      if(onOff == enabled) {
        dat <- dataIn.get_data();
        outDataAvailable(e);
      }
    }
    any dataOut.get_data() {
      return dat;
    }
    OnOffMode modeChange.get_modeVar() {
      return onOff;
    }
    void modeChange.set_modeVar(OnOffMode m) {
      onOff := m;
    }
  }
}

```

Fig. 4. Cadena Property Specification (CPS) (excerpts)

(e.g. `Timer` [20]) that is published by the RT event-channel itself (the RT-event channel contains dedicated timer threads to publish such events). More details of the event-channel threading model are given in the following section. Second, Bold Stroke applications follow in most cases a *control-push data-pull* architecture in which data is transferred between components in a two step process. First, a data producer such as `GPS` publishes a `DataAvailable` event indicating that it has updated some data that is ready to be consumed. Then, when a subscribing data consumer such as `AirFrame` receives the event, it calls an accessor method in a facet provided by the supplier (e.g., `dataOut` of type `ReadData`) to retrieve the data. Thus, threads never block waiting for data to become available, and this simplifies the design of real-time aspects. Note that under this strategy, component connections come in pairs consisting of an asynchronous event connection for notification and a synchronous interface/method connection for fetching the data.

3.3 Cadena system specifications

Thus far in the presentation of our example system, the only specification artifact that has been used is CCM IDL which specifies the interfaces of components. To generate system models appropriate for model-checking, additional specification forms are needed. Following our modeling strategy presented in Section 2, we need transition system specifications of components and specification of connection information – we address each of these in turn.

First, CCM does not include any mechanism for giving a high-level description of the behavior of components. Therefore, in Cadena we add a *component property specification* (CPS) format that allows developers to state several different light-weight semantic properties of components including mode-aware dependency specifications and transition system descriptions.

Figure 4 presents the transition system portion of the CPS description for the `LazyActive` and for the `Modal1` component type of Figure 2. The `LazyActive` component type implements a variant of control-push data-pull strategy to handle situations where a component C (e.g., `AirFrame` of Figure 3) depends on

data that is updated much more frequently than C 's clients require C 's data. For instance, in the example system of Figure 3, the `AirFrame` component does not fetch data immediately from `GPS` when notified that `GPS`'s data is available, but instead simply sets its `dataStatus` attribute to indicate that its data is stale (i. e. it was calculated based on `GPS` data that is now obsolete) and notifies its clients (e.g., `TacticalSteering`) that its data is available. It then retrieves and calculates the new data from the `GPS` only when it is actually ordered by one of the clients via its facet `dataOut`. In the CPS this behavior is captured by a mode variable `status` which links to the `dataStatus` attribute declared in the IDL. The behavior part provides a coarse description of the incoming ports' methods, i. e. in this example the handler method for an incoming `DataAvailable` event on port `inDataAvailable` (called the push-method in CORBA terminology), as well as the interface method `get_data` for the facet port `dataOut` (which is the accessor method for the `data` attribute of the `ReadData` interface). When the handler method receives an event, it sets the mode to stale and publishes an event itself. The interface method first checks on the mode and updates the internal data of the component accordingly before returning the value. If the value is updated, the mode is reset to fresh, so that in subsequent calls from the clients the data can be returned without retrieving new input via the `dataIn` receptacle again.

Note that in a real avionics system, there would be significant numeric computation to transform raw `GPS` data into a form that is useful for other components such as `AirFrame`. We do not represent this computation in our model for several significant reasons. First, in the actual systems supplied to us by Boeing, all such computation is stripped out for security reasons and to avoid dissemination of propriety information. Second, Boeing engineers are primarily concerned with reasoning about control properties associated with modes, and the data computations that are stripped out almost never influence the modal behavior of the system. In essence, Boeing engineers have by happenstance performed a manual abstraction of the system – an abstraction that produces a system that is very well-suited for model-checking in that remaining mode data domains are finite and small.

We represent the lack of concrete information about such stripped out data values by representing the associated variables with the CORBA type `any` (the top value in the CORBA type hierarchy). In addition, a statement such as `dat <- dataIn.getData()`; is not an assignment statement. Rather, it declares a dataflow dependency between `dat` and `dataIn.getData()`; which abstracts a situation where a series of assignments or method calls in the actual code transforms raw data received on the `dataIn` port to a refined value held in the `dat` variable. These dependency declarations are used in other components of Cadena, but for model-checking, they are left out of generated models since they have no influence on properties being checked.

Both `NavSteering` and `TacticalSteering` are *modal components* that have two modes (enabled, disabled). These modes are set by `PilotControl` via `ChangeMode` facets provided by the modal components. Figure 4 shows the `Modal1`, which is the type of the `TacticalSteering` component of Figure 3. When a modal component is disabled, any events received are simply discarded by the component. When enabled, the component responds according to the control-push data-pull

```

system ModalSPScenario {
  import cadena.common, cadena.modalsp;

  Rates 1, 5, 20;           // Hz rate groups
  Locations l1, l2, l3;    // abstract deployment locations
  ...
  Instance AirFrame implements LazyActive {
    connect this.inDataAvailable
      to GPS.outDataAvailable runRate 20;
    connect this.dataIn to GPS.dataOut;
  }
  Instance TacticalSteering implements Modal1 on l2 {
    connect this.inDataAvailable
      to AirFrame.outDataAvailable runRate 5;
    connect this.dataIn to AirFrame.dataOut;
  }
  ...
}

```

Fig. 5. Cadena Assembly Description for ModalSP (excerpts)

strategy (e.g., `TacticalSteering` responds to a `DataAvailable` from `AirFrame` by calling `AirFrame`'s `get data` method).

Having introduced the CPS format for specifying component behaviors, we now turn to the Cadena Assembly Description (CAD) format for specifying component instance allocations and connection information. While CCM allows components to be dynamically created and (dis)connected, Bold Stroke applications follow typical practice in safety/mission-critical systems and employ a static component allocation and configuration policy by creating and connecting components only in a system initialization phase. The CORBA 3.0 specification does not provide a dedicated language for static system configuration. Instead, an XML-based component assembly description is specified, leaving tool developers free to build a variety of configuration facilities which produce the XML data. Cadena provides graphical, textual, or form-based incremental static configuration facilities with the abstract syntax tree of the textual form providing the canonical representation. Figure 5 displays a fragment of the textual Cadena Assembly Description (CAD) for the example system. In CAD, a developer declares the component instances that form a system, along with event channel rate groups and abstract distribution locations. For receptacle and event sink ports, a `connect` clause declares a connection between a port of the current instance and a port of the component that provides the interface/event. This follows a convention that connections are declared on the client-side of an interface/event connection. Each event sink port connection uses the `runRate` clause to indicate which rate group thread should run the event handler upon event dispatch (thread rate groups are explained in the following section). A type-checking procedure ensures well-typed connections.

4 Real-Time Event Channel

In this section, we give a more detailed description of the CORBA real-time event channel and its elements as shown in Figure 6. This description will be used as a basis of explaining the Bogor event-channel models that Cadena uses when model-checking Bold Stroke systems.

In Bold Stroke applications, even though at a conceptual level component event source ports are connected to event sink ports, in the implementation,

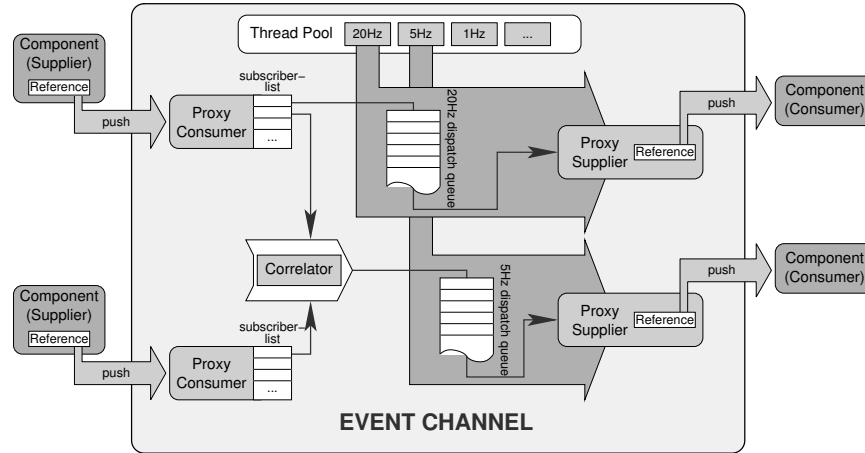


Fig. 6. RT CORBA Event Channel

event communication is factored through a real-time CORBA event channel. Use of such infrastructure is central to Bold Stroke computation because it provides not only a mechanism for communicating events, but also a pool-based threading model, time-triggered periodic events, and event correlation. In order to shield application components from the physical aspects of the system, for product-line flexibility, and for run-time efficiency, all components are *passive* – component methods are run by event-channel threads that dispatch events by calling the event handlers (“*push methods*” in CORBA terminology) associated with event sink ports. Thus, the event channel layer is the engine of the system in the sense that the threads from its pool drive all the computation of the system.

As defined in the CORBA standard an event connection consists of two types of objects, one being the *supplier* (i. e. an event source port), the other the *consumer* (i. e. the event sink port). An object of type consumer must provide a push method, i. e. the event handler method, which takes the event as argument. An object of type supplier stores a reference to a push method. To connect a supplier to a consumer, the supplier’s reference is set to point to the consumer’s push method. To publish an event e , the supplier simply calls the push method via its reference with the event e as argument (i. e. it “pushes” the event e). Complying to that scheme, the Event Channel offers a *proxy consumer*, i. e. a push method for each supplier to connect to. Similarly, for every consumer the Event Channel provides a *proxy supplier*, a reference to point to the consumer’s push method.

This simple reference-to-method pattern allows only one-to-one connections. Since more than one consumer might be interested in the events published by one supplier, the proxy consumer inside the Event Channel features a list of consumers to which an event originating from that supplier will be pushed. This list is called the *subscriber list*, as the consumers *subscribe* to the events of the supplier. This way every consumer and supplier only needs to handle one-to-one connections with the Event Channel, while a multiplexing of the events is done inside the channel.

The Event Channel also provides event correlation and event filtering mechanisms. In the example system of Figure 3, *and*-correlation is used, for instance,

to combine event flows from `NavSteering` and `AirFrame` into `Display`. The semantics of *and*-correlation on two events e_1 and e_2 is that the event channel waits for an instance of both e_1 and e_2 to be published before creating a notification event that is dispatched to the consumer of the correlation. The semantics of a correlator is defined by an automaton over event traces derived from the correlation expression [23].

The thread-pool shown in Figure 6 contains the three threads necessary to support the rate groups 20 Hz, 5 Hz, and 1 Hz of the example system of Figure 3. Following rate-monotonic theory, the 20 Hz thread has highest priority, followed by the 5 Hz thread and then the 1 Hz thread. There is an *event dispatch queue* associated with each thread t_r that holds pairs (s, e) where e is the event to be dispatched and s is the reference for an event sink port that is subscribed to the event. Thread t_r dispatches an event e from its queue by running the push method associated with port s via the reference in the related proxy supplier with the event e passed as a parameter.

Periodic computation is initiated by time-triggered events e_r for each rate r (e.g., events associated with event sinks of `Navigator`, `GPS`, and `PilotControl`). At the specified rate r (e.g., at 20 Hz for the `GPS` event sink), a special internal timer thread (not displayed) places a pair (s, e_r) in r 's queue for each component port s that subscribes to the timeout event e_r . Thread t_r dispatches these events by calling the push methods of subscribers via their proxy suppliers, which in turn may execute methods calls and publish other events to drive further computation.

There are three different paths through the event channel that e can take on its way to a particular subscriber s_k . First, in the normal path, the proxy consumer obtains the reference for s_k and the rate/priority r declared for s_k 's handler for e (recall from the discussion of Figure 5 that each non-time-triggered event sink port also has a rate identifier specified at configuration time that indicates which pool thread should be used to dispatch the event to which it is subscribed) from its subscriber list and puts the pair (s_k, e) in the queue for t_r .

Second, if s_k has an event correlation associated with e , the pair (s_k, e) is *not* placed in the queue. Rather, the correlator state machine is advanced to account for the publishing of e . If the correlator reaches an accepting state, then a pair (s_k, e_c) is placed in the queue matches the rate declared for s_k 's handler where e_c is a *correlation result event* possibly combines information from one or more events that were correlated.

The third path is an optimization path that short-cuts several steps in the event dispatch process based on the following observation: if there is no correlation associated with (s_k, e) and if subscriber s_k 's handler for e is declared to have the same rate/priority r as the thread t_r that is running e 's publisher, then (s_k, e) will be immediately placed in r 's dispatch queue and the same thread t_r will end up dispatching e . In this case, RT event channel implementation optimizes by having t_r directly call the push method for s_k with e as a parameter – thus, bypassing the queuing/dequeuing of (s_k, e) .

In detail, a trace using example of Figure 3 considering the 20 Hz thread would look as follows. A system interrupt causes the event channel's special timer thread to place a 20 Hz timeout pair (s_k, e_{20}) in the 20 Hz rate group dispatch queue for each 20 Hz time subscriber s_k (in this case, the only sub-

scriber is the `timeout` port of `GPS`. Since the 20 Hz queue is no longer empty, the 20 Hz-rate-group thread is started to call the event handler (push method) of `GPS`. Running the `GPS` handler for the timeout event reads data from the physical `GPS` device and issues a `DataAvailable` event, i. e. it calls the according push method in the connected proxy consumer inside the Event Channel. This method then, still executed by the same thread t_{20} , would typically queue the event into the dispatch queues of the subscribing components' thread groups. The subscriber for this `DataAvailable` event from the `GPS` is the `AirFrame`, which also belongs to the 20 Hz thread group, so in this case the optimization path is used and t_{20} thread directly calls the `inDataAvailable` event handler of `AirFrame`. This handler itself pushes a `DataAvailable` event the consumer proxy associated with the `AirFrame` `outDataAvailable` port. This proxy has a longer list of subscribers: it queues the event into the dispatch queue for the `NavSteering` and for the `TacticalSteering` component, and it forwards the event to the *and*-correlators which also consume events from `NavSteering` and `TacticalSteering` respectively. The state change in the correlators which reflects the incoming event is also executed by the 20 Hz thread, and so is the potential queueing of the correlated event into the `Display`'s rate group's dispatch queue. Since all of these components also run at 20 Hz, the according events are now found in the 20 Hz dispatch queue, and the 20 Hz thread will continue to execute. Assuming that the `TacticalSteering` component is enabled, while the `NavSteering` component is disabled, the push method which handles the incoming event for the `NavSteering` component simply ignores the event, while the `TacticalSteering` calls the `AirFrame`'s facet to fetch the newly available data. Upon this call, the `AirFrame` itself fetches the data from the `GPS`, turns over to fresh-mode and returns the data. After receiving the updated values, `TacticalSteering` issues a `DataAvailable` event itself. Its proxy now forwards this event to the correlator, which already is in a state indicating that the `AirFrame` has already sent his event, so that now a correlated event is queued for the `Display`. Again in the 20 Hz group this event is the last one in the queue executed by the thread. The thread runs the push method of the `Display`, which calls the facet of the `TacticalSteering` and receives the new data, and calls the facet of the `AirFrame`, which is in fresh-mode now so that it also immediately returns the new data. The data then is processed and displayed, and the 20 Hz thread ends until the next 20 Hz timeout.

5 Behavioral Models of Cadena Assemblies

5.1 Representing component structure and connections

As noted in Section 3, connections in current Bold Stroke systems are established in an initialization phase, and then remain fixed throughout the lifetime of the system. This means that although connection information must be represented, it does not need to be stored in the state vector. Similarly the interpretation of Cadena models in Bogor can be seen as two phases: first a buildup phase establishes the static part of the system in a single atomic step, then the connected system is checked over the state vector discussed below.

```

CAD.Component TacticalSteering;
enum EnabledDisabled { ENABLED, DISABLED }
EnabledDisabled tacticalSteeringMode;

TacticalSteering := CAD.createComponent(" TacticalSteering");
tacticalSteeringMode := EnabledDisabled.ENABLED;
CAD.declareEventSourcePort<EventType>(TacticalSteering, " outDataAvailable",
CAD.declareEventSinkPort<EventType>(TacticalSteering, " inDataAvailable",
                                     EventType.DataAvailable);
CAD.createField<Data>(TacticalSteering, " ReadData.data");
CAD.bindHandler<EventHandlerEnum>
  (EventHandlerEnum.tacticalSteering_push_inDataAvailable, TacticalSteering,
   "inDataAvailable");
...
CAD.connectEvent(AirFrame, " outDataAvailable",
                 TacticalSteering, " inDataAvailable", 20, false);
...

function tacticalSteering_push_inDataAvailable(
    EventHandlerEnum eh, CAD.Event event) {
  Data dat;
  loc loc0: live {}
  when (onOff == EnabledDisabled.ENABLED) do { } goto loc1;
  when !(onOff == EnabledDisabled.ENABLED) do { } return;
  loc loc1: live {} invisible invoke airFrame_facet() goto loc2;
  loc loc2: live {dat}
  when true do {
    dat := CAD.getField<Data>(AirFrame, " ReadData.data");
  } goto loc3;
  loc loc3: live {}
  when true do {
    CAD.setField<Data>(TacticalSteering, " ReadData.data", dat);
  } goto loc4;
  loc loc4: live {}
  invisible invoke pushOfProxy(TacticalSteering, " outDataAvailable", ...);
  return;
}
...

```

Fig. 7. Bogor component and assembly descriptions for ModalSP (excerpts)

The buildup phase begins with the creation of component instances followed by actions that connect the ports of each instance to ports of other instances (in the case of facets/ receptacles) or the model of the real-time event-channel (in the case of event source/sinks). Figure 7 shows how the Bogor CAD extension supports the buildup of data structures representing components. This extension (not shown) declares two new types `Event` and `Component` (which is used as the type of the BIR `TacticalSteering` variable). Further, the extension defines a number of operations such as `createComponent` and `declareEventSourcePort` that are implemented by Java methods in the extension. Note for example the use of the `bindHandler` operation which declares that the BIR function displayed at the bottom of Figure 7 is to be used as the event handler for events flowing into the `inDataAvailable` port of `TacticalSteering`.

Below the declaration of the component structure, Figure 7 illustrates the use of the `connectEvent` method. This action causes a Bogor reference to the `inDataAvailable` port of `TacticalSteering` to be added to the subscriber list (recall the discussion of Figure 6 in Section 4) for the `outDataAvailable` port of `AirFrame`.

When implementing a Bogor extension, one must define a *state manager* that walks over the extension's state and produces a representation suitable for placing in the model-checker's state vector. This flexibility can be leveraged

```

Queue.type<Pair.type<EventHandlerEnum , CAD.Event>> Q5;
...
Q5 := Queue.create <...>(MaxCapacity.QUEUE);
CAD.bindDispatchingQueue <...>(Q5, 5);
...
thread threadgroup5() {
  Pair.type<EventHandlerEnum , CAD.Event> pair;
  EventHandlerEnum handler;
  CAD.Event event;

  loc loc0: live { handler , event } when Queue.size <...>(Q5) > 0
  do invisible {
    pair := Queue.getFront <...>(Q5);
    Queue.dequeue <...>(Q5);
    handler := Pair.first <...>(pair);
    event := Pair.second <...>(pair);
  } goto loc1;
  loc loc1: live {}
  invisible invoke virtual f(handler , event) goto loc0;
}

```

Fig. 8. Bogor dispatch queue and thread model for ModalSP (excerpts)

in a variety of ways, e.g., to omit various fields from the data vector, to form abstractions of the state, or to build canonical representations necessary for achieving symmetry reductions in the representations of sets [21]. In Cadena models, we use this mechanism to avoid storing the static connection information in the state vector. This also allows us to increase the granularity of actions in initialization and in middleware actions – thus, soundly reducing the number of interleavings. Moreover, traversal of subscriber lists can sometimes be carried out atomically (depending on priorities of threads involved), since there is no chance of interfering updates to subscriber lists once execution begins.

5.2 Representing component behavior

Event handlers and other methods of CCM components are represented as BIR functions. Figure 7 shows the BIR model of the event handler for the `indataAvailable` event sink from `Modal1` component type as defined in CPS definition of Figure 4. The transitions capture the handler behavior as defined in the CPS file: if the component is disabled, the handler simply returns, otherwise it fetches data using its `dataIn` port, updates its local data, and then publishes a `dataAvailable` event on its `outDataAvailable` port.

In the example Bold Stroke systems supplied by Boeing, the concrete internal data of components consists exclusively of the values of component mode variables (e.g. the `enabled/ disabled` values of mode variable `onOff` from component `Modal1`, Figure 4). As discussed in Section 3, Boeing engineers abstract away the other data values such as the actual numerical data produced by e.g. GPS devices. Thus, such values are represented by a BIR extension type `Data` (as equivalently by the type `any` in the CPS, Figure 4) that has a single dummy value. Using the state representation mechanism introduced above, component fields of type `Data` are not held in the state vector. This means that component models only contribute the values of mode variables to the state vector.

5.3 Representing the real-time event channel

The BIR model of the real-time CORBA event service represents the thread-pool, event dispatch queues, and correlators presented in Figure 6 of Section 4. Recall from Section 4 that dispatch queues hold event/subscriber pairs (s, e) . In the Bogor model, queues are modeled using `Queue` and `Pair` extensions. Figure 8 illustrates the 5 hertz rate queue of pending event dispatches and the thread, `threadgroup5`, that cyclically dequeues dispatch pairs and invokes the component event handler encoded in each pair (note that pair type declarations are elided (i.e., $\langle \dots \rangle$) for improved readability).

Each correlator is represented as a deterministic finite-state automaton whose transition function is encoded as a static transition table. For each correlator, there is a single state variable that holds the current correlator state. Since the structure of correlators is fixed for a given system, the transition tables are not held in the state vector.

5.4 Summary of data portion of state-vector

To summarize the modeling strategy discussed above, we present the state vector components related to data state of Cadena systems. The *observable state* of a Cadena assembly is comprised of all non-fixed system data. As we have noted above, correlator transition tables, subscriber lists, and component connection information are all fixed and are not considered part of the observable state.

Definition 1. *Cadena Data States are tuples $(\mathbf{c}, \mathbf{r}, \mathbf{a}, t, p)$ where:*

$\mathbf{c} = (c_1, \dots, c_k)$ stores the data states of component instances, each of which is comprised of a, possibly empty, set of mode attributes as defined by c_i 's component type.

$\mathbf{r} = (q_{r_1}, \dots, q_{r_n})$ are rate-specific queues of pairs, (c, e) , recording the dispatch of event e to component c .

$\mathbf{a} = (a_1, \dots, a_l)$ stores the current states of each of the event correlation recognition automata.

t records an abstraction of time used to trigger timeouts.

p records the priority of the current thread being executed.

The initial state is defined to have instance modes set to their initial values, correlation automata set to their start state, rate specific queues to be empty, $t = 0$, and the priority variable is set to the highest priority.

In addition, the values of local variables in component handlers and methods and in the implementation of push methods and rate-specific threads cannot be observed outside their method activation by other threads or by property observables and are also not considered part of the observable state. Local variable *are* held in the state vector, but only during the corresponding method activations.

5.5 Strategies for modeling scheduling and time

The behavior of Cadena systems is driven by the triggering of middleware timeouts as described in Section 4 and is controlled by the scheduling policies of the

thread-pool in the real-time event channel. Finding an effective strategy for modeling these timeouts and thread-scheduling is a central issue in the construction of Cadena models.

When analyzing concurrent systems, most model-checkers do not attempt to exploit knowledge of specific timing or scheduling strategies but instead explore all possible interleavings of concurrent actions. If we followed this approach, we would allow timeout events to occur non-deterministically between every system transition and we would allow actions from different threads to be interleaved non-deterministically without consideration of priorities or other scheduling constraints. While such a strategy is *sound* in that it covers all possible system behaviors, the number of states generated makes it impractical for all but the smallest systems.

In the subsections below, we describe several strategies that we use to reduce infeasible interleavings. Each strategy incorporates constraints based on observations about priority scheduling and timeout policies implemented by the real-time middleware.

Priority-based scheduling: Having the model-checker non-deterministically explore interleavings without considering thread priorities obviously introduces schedules that are infeasible in the actual system, e.g., a schedule that continues to execute transitions from a lower priority thread even though a higher-priority thread is enabled.

Inter-rate-group timeout constraints: Having the model-checker non-deterministically generate timeout events introduces schedules that are infeasible in the actual system, e.g., a 5 Hz timeout event should not occur more frequently than a 20 Hz timeout event. We present strategies that reduce infeasible interleavings by taking into account the appropriate relative frequency of timeout events, i.e., by taking into account constraints that exist between timeouts of different rate groups.

Intra-rate-group timeout constraints: Having the model-checker non-deterministically generate timeout events introduces infeasible schedules where a timeout for a rate group r occurs before all events in the current frame for r are dispatched or before the previous timeout from group r is even dispatched. We constrain the generation of time-out events to ensure that timeouts from the same rate group are not triggered “too quickly”.

This strategy constrains the occurrence of timeouts by considering the relative lengths of the real-times frames and constrains scheduling by considering priority information.

Lazy-time with priority scheduling: In addition to the techniques used in the strategy above, this strategy also considers timing estimates for system transition which allows additional infeasible schedules to be removed from consideration.

5.6 Representing priority-based scheduling information

Bold Stroke systems are priority scheduled based on the results of rate monotonic analysis of a set of harmonic rate groups. The CAD call `connectEvent()`, illustrated in Figure 7, assigns a rate, and hence a priority, to each component

handler for a given event. The default non-deterministic scheduling policy in Bogor is implemented by a module that calculates the set of enabled transitions in a given state and passes that set to the state exploration module, which explores each possible outgoing transition. When reporting our experiments, we refer to models that use this strategy as *priority-unaware*. For Cadena models, a Bogor plugin is used that intercepts the set of enabled transitions in a given state, selects the transition with the highest priority and passes that single transition on to the state exploration module. As expected, this yields dramatic reductions in the state space, as shown in Section 6, and also improves the precision of the state space since only infeasible schedules are eliminated (i.e., ones on which a lower-priority transition executes when a higher-priority transition is enabled). We refer to models that use this strategy as *priority aware*. Variations of this plugin are used in the following models to allow for interleaving of timeouts with the highest-priority enabled transition.

5.7 Representing intra-rate-group timing constraints

The treatment of time, t , determines, in part, the fidelity of the model with respect to the real system’s behaviors. If detailed timing information is available one can keep track of time as component actions are executed and use that time value to trigger periodic events. However, even when timing information is not available, one can still reduce the occurrence of timeout events based on both intra- and inter-rate-group constraints.

Intra-rate-group constraints that we consider involve the notion of *frame overrun*. A frame overrun occurs when a timeout event e_r for rate group r occurs before all events e' triggered directly or indirectly by the previous timeout for r are processed by the rate group’s thread t_r . In normal situations, a timeout e_r occurs and is dispatched, other events arrive in the event channel’s dispatch queues (including those associated with r), and thread t_r becomes idle after all events associated with r have been dispatched. The time that t_r remains idle waiting for the next r timeout is called *slack time*. If a system has a frame overrun error, a thread t_r has no slack time – it is unable to finish all of its work before the next timeout e_r arrives.

Note that exploring the state-space of systems where arbitrary frame overruns are modeled results in a huge number of additional system behaviors that would very likely be infeasible if actual timing data were considered (timing data would allow us to conclude that in most cases frame overruns do not occur). While frame overruns are a real source of bugs in Bold Stroke systems, engineers have other tools and methods for detecting these types of errors. Accordingly, we will reduce the state space that we explore using two strategies. The first strategy which we call **no overruns** assumes that no frame overruns occur at all. This is implemented by having the model-checker scheduler only emit a timeout event for rate group r if there are no enabled transitions associated with rate group r – which models the situation where t_r has become idle. The second strategy which we call **limited overruns** is implemented by having the model-checker scheduler only emit a timeout event e_r if there is no other timeout event remaining in the r dispatch queue (but other non-timeout events may still be waiting in the queue for dispatch). Intuitively, that this model includes overruns that only spill over

into the very next frame but does not include overruns where processing is 'late' by more than one additional frame.

5.8 Representing inter-rate-group timing constraints

The strategies related to buffer overrun in the previous section constrain timeout events by considering when they should occur relative to other timeouts from the *same* rate group. We now a strategy which we call the **relative-time (RT)** strategy that constrains the issuing of timeout events by considering when a timeout for r should occur relative to a timeout for a *different* rate group r' . Specifically, we take advantage of the fact that in rate-monotonic scheduling theory (which is used in Bold Stroke systems), the frame associated with a rate can be evenly divided into some whole number of r' -frames for each rate r' that is higher than r . In the example system of Figure 3, the frame of the slowest rate (1 Hz) can be divided into 5 Hz frames, and each 5 Hz frame can be divided into 4 20 Hz frames. The longest frame/period (the frame associated with the lowest rate) is called the *hyper-period*.

In general, using priority scheduled models and assuming the **no overruns** strategy from above, the **relative-time** model enforces the following constraints related to issuing of timeouts:

- a single timeout is issued for the slowest rate group in the hyper-period,
- timeouts for rate groups, r_i and r_j where $r_i > r_j$, are issued such that r_i/r_j timeouts of rate r_i are issued in a r_j frame.

These constraints determine the total number and relative ordering of instances of timeouts that may occur in the hyper-period.

Figure 9 shows the Bogor code for two threads that are used to model this strategy. Thread `timerThread` increments an abstraction of time where each 'tick' (i.e., each increment of the `time` variable) represents the passing of time corresponding to the shortest frame in the system (e.g., in the ModalSP, each tick represents a 20 Hz frame). The `time` variable wraps around every 20 ticks which corresponds to the fact that there are 20 Hz frames in the 1 Hz hyper-period. Thread `timeOutSenderThread` models the behavior of the rate-specific timer threads in the middleware discussed in Section 4. This thread monitors `time` and when it observes a change in the time value, it passes through a `case` statement to see which timeout events should be dispatched at that point. Since a `time` tick represents the period of the shortest frame, a new timeout event for the fastest rate is issued on each pass through the `case` statement. In our example system, the 5 Hz timeout happens every fourth tick. To represent the occurrence of a timeout, the thread enqueues the timeout event through the standard push call.

From the explanation above, it is clear that the **RT** model only establishes the occurrence of timeouts relative to each other – it does not relate timeout occurrences to the time required by component event handlers and method execution. Thus, it is now important to understand when timeout actions may occur with respect to actions that occur inside of component handlers, i.e., when can these actions be interrupted by timeouts.

```

CAD.Component Timer;
...
Timer := CAD.createComponent("Timer");
CAD.declareEventSourcePort<EventType>(Timer,"timeOut5",EventType.TimeOut);
...
thread timerThread() {
  loc loc0: live {}
  when true do { time := (time + 1) % 20; } goto loc0;
}
...
thread timeOutSenderThread() {
  ...
  loc loc1: // 5 Hz timeout case
  when time % (20/5) == 0 do invisible {} goto locInvoke5;
  when time % (20/5) != 0 do invisible {} goto loc2;
  ...
  loc locInvoke5: live {localTime}
  invisible invoke pushOfProxy(Timer, "timeOut5",
                              CAD.createEvent<EventType>(EventType.TimeOut))
  goto loc2;
  ...
  loc loc2: // 1 Hz timeout case
  ...
}

```

Fig. 9. Timer and TimeOutSender thread models for ModalSP (excerpts)

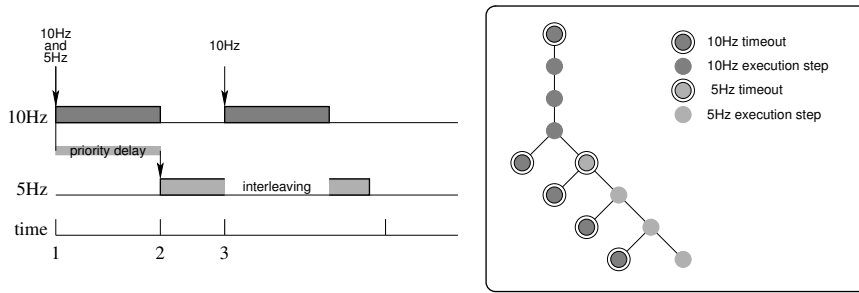


Fig. 10. Relative-time Environment

To see that the model safely approximates all interleavings of timeouts and component actions (given the constraint on no frame overruns) consider Figure 10. This figure illustrates four points during a system execution which contains 5 Hz and 10 Hz rate processing. The 10 and 5 Hz timeouts are queued together (e.g., at the point 1) since they both have frames that begin at the same point. However, the 10 Hz timeout event is dispatched first due to its higher priority. Once the all the actions associated with 10 Hz component processing complete (e.g., at point 2), the model-checker scheduler begins consideration of lower priority actions and the 5 Hz timeout is dispatched leading to 5 Hz component processing. Our **no overruns** assumption entails that processing the 10 Hz component actions does not require more time than the period of the 10 Hz frame — thus, the next 10 Hz timeout cannot occur before point 2. Since we are not modeling the actual time required for carrying out component actions, it impossible to determine the relationship between the time required for 5 Hz component action processing (e.g., the duration from point 2 to 3) and the time until the next 10 Hz timeout (e.g., the duration from point 2 to 4). To safely cover all possibilities, we must allow for any relationship between these durations. To model all such relationships, we adapt Bogor’s standard scheduling

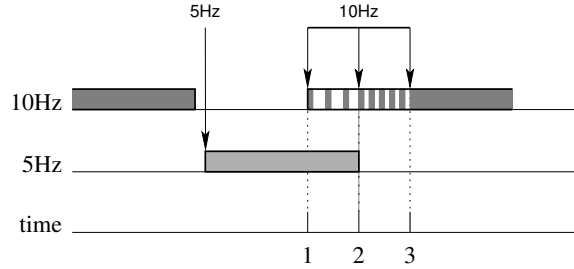


Fig. 11. Lazily-timed Environment

module to consider all interleavings of enabled timeouts with the enabled transitions. On the right in Figure 10 the interleavings of the 10 Hz timeout and the enabled transitions performed during 5 Hz component processing are illustrated. The first white circle represents the dispatching of the leftmost 10 Hz timeout event. This is followed by black circles representing transitions in 10 Hz component processing; the first branch point represents the choice between the next 10 Hz timeout (on the left) or dispatching the already queued 5 Hz timeout event (on the right). If 5 Hz processing is selected then the choice between the 10 Hz timeout and 5 Hz processing repeats for the next enabled 5 Hz transition illustrated as a grey circle.

5.9 Lazily-Timed Components

In the **relative-time** model, timeouts are arranged in a proper order and ratio with respect to each other, but there are no constraints that guarantee that, e.g., the interval between time outs is appropriate for the correspond period. This means that the model may have interleavings in which a timeout, e.g., for r_i , occurs prematurely with respect to an action sequence whose duration is less than $period(r_i)$. For example, if the 5 Hz component processing (i.e., from point 2 to 3) in Figure 10 is guaranteed to be less than the time to the next 10 Hz timeout (i.e., from point 2 to 4) then the interleavings of timeouts with 5 Hz processing in the *RT* model will be infeasible. The **lazily-timed (LT)** component model addresses this by leveraging worst-case estimates of the running time of components; these will be available for Cadena systems to support rate monotonic analysis. This model can be configured for whatever granularity of timing information is available. Here we consider worst-case timing estimates for event handlers. Conceptually, the estimates are used to determine whether a handler can run without interruption before the next timeout occurs and, if not, the model non-deterministically interleaves action sequences from the handler with timeouts and higher-priority actions that follow from timeouts.

This model modifies the data associated with time to record the intra-hyperperiod (IHP) time normalized by the least common factor of all handler durations and timeout periods, the guards in `timeOutSenderThread` from Figure 9 are adjusted accordingly, and each component handler is modified to include an increment of time. Figure 11 illustrates how these increments are performed. It shows the execution of 5 Hz component processing subsequent to completion of 10 Hz processing in a frame. There are two cases: (1) the worst-case time estimate of the 5 Hz processing (i.e., which runs up to point 2) is less than or

equal to the next timeout (i.e., timeout occurs at point 3) or (2) it is not (i.e., timeout occurs at point 1 and interrupts the 5 Hz actions). In case (1), the IHP time is incremented by the worst-case timing estimate of the currently running 5 Hz event handler and the state space exploration algorithm proceeds; note that there is no branching in the state space for this case. In case (2), the IHP time is incremented to the next timeout (i.e., point 1), a non-deterministically chosen prefix of the currently running 5 Hz handler is executed, and then the 10 Hz timeout is performed. By choosing a prefix of the handler actions, we are modeling all possible distributions of timing across the actions of the handler. The remaining portion of the handler is left for the state-space exploration algorithm after the 10 Hz timeout, and subsequent 10 hertz processing is performed. The difference between point 3 and point 2 (i.e., the worst-case execution time of the handler) is assigned to that remaining portion as its duration.

This model can be seen as a refinement of the **RT** model. It eliminates interleavings when the timing estimates guarantee that a group of highest-priority enabled transitions are guaranteed to complete before the next timeout. In the example in Figure 10, if the right-most three light-grey circles correspond to a 5 Hz component handler body whose worst-case execution bound is less than the time to the next 10 Hz timeout, then there would be no branching in that portion of the state space (i.e., the lower two left outgoing arcs to 10 Hz timeouts are eliminated).

6 Experimental Results

Table 1 shows the results of evaluating our strategies using four example systems provided by Boeing engineers. As an example of how to read to system description, the **ModalSP** scenario that we have used as an example has three threads (for rate groups 1 Hz, 5 Hz, and 20 Hz), 8 components, an event correlation (e/c), and 125 events being generated per one second hyper-period (hp).

For each scenario, we give data for five models that incorporate the modeling strategies presented in the previous section.

- (**R**) is the reference model. There is no scheduling policy for the thread groups in the scenario (it is **priority unaware** and has no intra- or inter-rate-group timing constraints). Thus, all job executions are interleaved.
- (**RT-1**) uses two policies: **priority aware** scheduling and the **relative time** environment where we implement the **no frame overruns** strategy for the highest-priority thread.
- (**RT-2**) is like (RT-1), but also assumes there are **no frame overruns** for all threads.
- (**LT**) is like (RT-2) but uses the **lazy time** environment model.

For each example, we collect the number of transitions *trans*, *states*, *time*, and memory consumption *mem* at the end of the search. The numbers of transitions and states are both listed because some of steps in the model are marked as invisible (atomic) for which Bogor will not save the states. The experiments were run on a Pentium 4 2.53GHz with 1.5Gb RAM using the Java 2 Platform.

Example System		(R)	(RT-1)	(RT-2)	(LT)
<i>Basic Scenario</i>	<i>trans</i>	111	42	42	44
Threads: 20Hz	<i>states</i>	20	12	12	14
Components: 3	<i>time</i>	.16 sec	.11 sec	.09 sec	.11 sec
Events: 2 per .05sec hp	<i>mem</i>	.51Mb	.5Mb	.5Mb	.51Mb
<i>Multi-Rate Scenario</i>	<i>trans</i>	1.36M	7.5K	.98K	.15K
Threads: 20Hz, 40Hz	<i>states</i>	.12M	1.5K	.1K	33
Components: 6	<i>time</i>	5 min	1.9 sec	.38 sec	.19 sec
Events: 6 per .05sec hp	<i>mem</i>	16Mb	.77Mb	.61Mb	.61Mb
<i>ModalSP Scenario</i>	<i>trans</i>	<i>o.m.</i>	.92M	38.2K	6.27K
Threads: 1Hz, 5Hz, 20Hz	<i>states</i>	3M+	20.9K	9.1K	1.56K
Components: 8 (e/c)	<i>time</i>	<i>o.m.</i>	20 sec	8.59 sec	2.11 sec
Events: 125 per 1sec hp	<i>mem</i>	<i>o.m.</i>	4.1Mb	1.61Mb	1.45Mb
<i>Medium Scenario</i>	<i>trans</i>	<i>o.m.</i>	<i>o.m.</i>	3.79M	.36M
Threads: 1Hz, 20Hz	<i>states</i>	—	13M+	.74M	74.5K
Components: 50	<i>time</i>	<i>o.m.</i>	<i>o.m.</i>	29 min	3 min
Events: 820 per 1sec hp	<i>mem</i>	<i>o.m.</i>	<i>o.m.</i>	71.8 Mb	21.5Mb

Table 1. Experiment Data

Bogor’s collapse compression [16] and heap symmetry [17] and process symmetry [2] reductions are used in all of the experiments. Each of the experiments represents a complete exploration of the state-space of the system.

From the table, the state space generally decreases from model (R), (RT-1), (RT-2), to (LT). This shows that by incorporating more knowledge (e.g., the scheduling policy) of the model that is being checked, less states need to be explored. For example, *Medium*, the largest scenario that we have, cannot be model checked using Bogor or our previous dSpin implementation [13] without employing the reduction strategies used in (RT-2) and (LT). For *Basic* the states are the same for model (RT-1) and (RT-2) because it only has a single thread (thus, there is no interleaving). Model (R) has a larger number of states because the lack of constraints allows the timeout to occur even when events associated with the current frame are still being dispatched. Model (LT) has two more states than (RT-2) due to the overhead introduced by the timing transitions.

Bogor runs out of memory checking *ModalSP* (R) (at 3 million states) and *Medium* (RT-1) (at 13 million states). It is interesting that the states for *ModalSP* (R) require more memory than the states for *Medium* (RT-1). This is an effect of the collapse compression that is used. Specifically, there are three threads in *ModalSP* (R), but only two threads in *Medium* (RT-1). In addition, *ModalSP* (R), which has fewer scheduling constraints, allows more interleaving than *Medium* (RT-1). Thus, the collapse compression can save more in *Medium* (RT-1) than *ModalSP* (R), because there are more similar state bit patterns in *Medium* (RT-1) than in *ModalSP* (R).

7 Related Work

Garlan and Khersonsky [11] describe an approach for checking publish-subscribe systems (which they refer to as implicit invocation systems) using SMV. We build on their key insights of factoring system models into two parts: (1) a reusable

model of run-time event-delivery infrastructure, and (2) application dependent, user-specified component models. However, we support much more directly the forms of component structure and connections (i.e., CCM structures and object references) and event-delivery mechanisms (i.e., RT CORBA middleware) found in real systems. This advance is achieved by leveraging Bogor’s direct modeling of OO concepts (as compared to the SMV input language which provides very little support for modeling programming language features) and Bogor’s extension mechanism (which allows complex middleware behavior to be captured internally to the model-checker). In addition, [11] does not provide any performance data, nor does that work consider any state-space reductions based on priorities, scheduling, or timing constraints that seem critical for scaling to realistic applications.

There has been a large body of work on timing and schedulability analysis for component-based systems. As these techniques have matured, they have been integrated into environments that support the development of real-time systems. For example, MetaH [24] and Geodesic [6] are frameworks that support the reuse of components written in Ada and Java, respectively, in real-time systems. These frameworks include a range of timing analyses and automatically generate infrastructure code that coordinates the execution of component code in a way that achieves the system’s timing requirements. Cadena is complementary to this work in that it targets logical properties of a system using both light-weight and heavy-weight analysis techniques.

Ptolemy [19] is a framework that allows a wide variety of formal descriptions of components and their behavior to be integrated into a single system. User’s provide sufficient detail in these descriptions to allow implementations to be automatically generated. Ptolemy provides a run-time infra-structure to mediate between components that have different execution models. In contrast, Cadena models intentionally leave out detail in order to provide more abstract system descriptions that are amenable to analysis for large systems. While Cadena provides some code generation capabilities, we do attempt to generate component method implementations.

Model checking [4] has become extremely popular as a technology for analyzing behavioral models of software artifacts. Researchers have extracted such models from source code (e.g., [14, 5]), UML design artifacts (e.g., [20, 18]) and architectural descriptions (e.g., [1]). The difficulty with all applications of model checking is scaling it up to apply to realistically large and complex systems. Recent years have seen an enormous amount of research on the systematic abstraction of models to enable more tractable reasoning. We take a different approach in Cadena by exploiting the *natural* abstractions that arise when developing high-level design models of systems.

8 Conclusions

We believe the idea of a flexible model-checking framework that allows domain-specific extensions (such as the ones that we have used for encoding models of CORBA communication layers) can be a very effective approach for model-checking modern distributed system designs and implementations. We are currently working with Boeing engineers to incorporate other forms of domain in-

formation (e.g., common specification idioms) and other forms of light-weight checking (e.g., interface protocol checking, refinement checking) and static analysis into Cadena.

References

1. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.
2. D. Bosnacki, D. Dams, and L. Holenderski. Symmetric spin. In *International Journal on Software Tools for Technology Transfer*. Springer-Verlag, 2002.
3. G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder – a second generation of a Java model-checker. In *Proceedings of the Workshop on Advances in Verification*, July 2000.
4. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
5. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera : Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.
6. D. de Niz and R. Rajkumar. Geodesic - a reusable component framework for embedded real-time systems. Technical report, Carnegie Mellon University, 2002.
7. C. Demartini, R. Iosif, and R. Sisto. dspin : A dynamic extension of SPIN. In *Theoretical and Applied Aspects of SPIN Model Checking (LNCS 1680)*, Sept. 1999.
8. W. Deng, M. Dwyer, J. Hatcliff, G. Jung, Robby, and G. Singh. Model-checking middleware-based event-driven real-time embedded software (extended version). Forthcoming – April 2003.
9. B. Doerr and D. Sharp. Freeing product line architectures from execution dependencies. In *Proceedings of the Software Technology Conference*, May 1999.
10. Eclipse Consortium. Eclipse website. <http://www.eclipse.org>, 2001.
11. D. Garlan and S. Khersonsky. Model checking implicit-invocation systems. In *Proceedings of the 10th International Workshop on Software Specification and Design*, Nov. 2000.
12. T. H. Harrison, D. L. Levine, and D. C. Schmidt. The design and performance of a real-time corba event service. In *Proceedings of the 1997 ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, pages 184–200. ACM Press, 1997.
13. J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad. Cadena: An integrated development, analysis, and verification environment for component-based systems. In *Proceedings of the 25th International Conference on Software Engineering (to appear)*, 2003.
14. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 1999.
15. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.
16. G. J. Holzmann. State compression in SPIN: Recursive indexing and compression training runs. In *Proceedings of Third International SPIN Workshop*, Apr. 1997.
17. R. Iosif. Symmetry reduction criteria for software model checking. In *Proceedings of Ninth International SPIN Workshop*, volume 2318 of *Lecture Notes in Computer Science*, pages 22–41. Springer-Verlag, Apr. 2002.
18. D. Latella, I. Majzik, and M. Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.

19. E. A. Lee. Overview of the ptolemy project. Technical Report UCB/ERL M01/11, University of California, Berkeley, Mar. 2001.
20. J. Lilius and I. P. Paltor. vUML: A tool for verifying UML models. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, 1999.
21. Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. Technical Report SANTOS-TR2003-3, Kansas State University, 2003. (submitted for publication).
22. Robby, M. B. Dwyer, and J. Hatcliff. Bogor Website. <http://www.cis.ksu.edu/bandera/bogor>, 2003.
23. H. Sipma. Event correlation: A formal approach. Technical Report Draft, Stanford University, July 2002.
24. S. Vestal. Metah user's manual. <http://www.htc.honeywell.com/metah>, 1998.