A Case Study in Domain-customized Model Checking for Real-time Component Software

Matthew Hoosier[1], John Hatcliff[1], Robby[1], and Matthew B. Dwyer[2]

[1] Kansas State University
Manhattan, KS 66506, USA
{matt,hatcliff,robby}@cis.ksu.edu
[2] University of Nebraska
Lincoln, NE 68588, USA
dwyer@cse.unl.edu

Abstract. Despite a decade of intensive research on general techniques for reducing the complexity of model checking, scalability remains the chief obstacle to its widespread adoption. Past experience has shown that domain-specific information can often be leveraged to obtain state-space reductions that go beyond general purpose reductions by customizing existing model checker implementations or by building new model-checking engines dedicated to a particular domain. Unfortunately, these strategies limit the dissemination of model checking across a number of domains since it is often infeasible for domain experts to build their own dedicated model checkers or to modify existing model checking engines.

To enable researchers to more easily tailor a model checking engine to a particular software-related domain, we have constructed an extensible and highly explicit-state software model checking framework called Bogor. In this paper, we describe our experience in customizing Bogor to check design models of avionics systems built using real-time CORBA component-based middleware. This includes modeling the semantics of a real-time CORBA event channel as a Bogor abstract data type, implementing a customized distributed state-space exploration algorithm that leverages the quasi-cyclic nature of periodic real-time computation, and encapsulating the Bogor checking engine in a robust full-featured development environment called Cadena that we have built for designing, analyzing, synthesizing, and implementing systems using the CORBA Component Model.

1 Introduction

Despite a decade of intensive research on general techniques for reducing the complexity of model checking (e.g., see [5]), scalability remains the chief obstacle to its wide-spread adoption. It is common-place for model checkers to exhaust available memory when analyzing even highly-abstract models of real systems. While general reduction methods are effective (e.g., partial order reductions can lead to order of magnitude space reductions), past experience of a number of researchers has shown that information about the *system domain* can be exploited to enable further significant space reductions. In some cases, a *new* model-checking framework was developed targeted to the semantics of a family of artifacts [3, 11], while in other cases it was necessary to study an *existing* model checking framework in detail in order to customize it [4, 6].

Unfortunately, this level of knowledge and effort currently prevents many *domain* experts who are not necessarily experts in model-checking from successfully

applying model checking to software analysis. Even though experts in different areas of software engineering have significant domain knowledge about the semantic primitives and properties of families of artifacts that could be brought to bear to produce cost-effective semantic reasoning via model checking, these experts should not be required to build their own model-checker or to pour over the details of an existing model-checker implementation while carrying out substantial modifications.

To enable researchers to more easily tailor a model-checking engine to a particular software-related domain, we have constructed an extensible and highly modular explicit-state model checking framework called Bogor [18].[1]

For treating realistic designs and implementations in widely-used languages such as Java and C#, Bogor, provides rich base modeling language including features that allow for dynamic creation of objects and threads, garbage collection, virtual method calls and exception handling. For these built-in features, Bogor employs state-of-the-art reduction techniques such as collapse compression [14], heap symmetry [15], thread symmetry [2], and partial-order reductions. For tailoring to specific domains, Bogor provides (a) mechanisms for extending Bogor's modeling language with new primitive types, commands, expressions, and 02 associated with a particular application, and (b) a well-organized module facility for plugging customized domain-tailored components into the model-checking engine. Moreover, Bogor is designed to be easily encapsulated within larger domain-specific development and verification environments.

In this paper, we report on a case study in which we used the above facilities to customize Bogor for checking properties of avionics system designs. In this domain, real-time event-driven systems are built from components defined in the CORBA Component Model (CCM) and deployed on Boeing's Bold Stroke middleware infrastructure built from a variety of real-time oriented services and the ACE/TAO real-time CORBA code base. Bogor is used to check global temporal properties of transition systems models formed from component transition systems as well as dedicated abstract domain models that capture the behavior of real-time middleware and services. Specifically,

- *new Bogor types* are defined to components, component ports, and events used for inter-component communication,
- *new Bogor modeling language commands and expressions* are introduced for declaring and connecting components and component ports, publishing and subscribing to events, and making calls on component method interfaces,
- *new Bogor internal modules* that model the complex semantics of the multi-layered ACE/TAO real-time event channel in which a pool of real-time threads is used to dispatch events to components and drive the computation of the system,
- *new state vector representations* are created that avoid storing data that does not change during execution in this domain (e.g., connection information for components),
- the general-purpose non-deterministic model-checker scheduler is replaced by a *new domain-specific scheduler* that implements the particular real-time rate-monotonic scheduling policy of the ACE/TAO real-time event channel of the Bold Stroke environment,
- the standard depth-first search algorithm of the model checker is replaced by a *distributed quasi-cyclic search algorithm* that takes advantage of the

[1] Bogor project website: `http://bogor.projects.cis.ksu.edu`

periodic nature of systems in this domain to achieve dramatic reductions in space and time.

This customized model checking framework has been encapsulated in a larger tool called Cadena[2] that we have constructed for supporting model-driven development and analysis of systems built from both industry standard component models such as the CORBA Component Model (CCM) and Enterprise Java Beans (EJB) as well as proprietary models such as Boeing's Prism component architecture. This required building translators to/from Cadena's design notations to Bogor's transition system notations.

In earlier work, we have described the design rationale, features, and implementation of Bogor [18] and Cadena [12], different approaches for modeling the ACE/TAO real-time event channel in Bogor [7], and the foundations of a sequential version of the quasi-cyclic search algorithm [9]. In this paper, we focus on our experience in carrying out the tasks involved in the overarching goal of customizing Bogor to check real-time designs as part of Cadena, and we highlight the incorporation of the *distributed version* of the quasi-cyclic search strategy which significantly increases the scalability of the approach.

Compared to previous work on model-checking publish-subscribe architectures and component-based systems, our approach breaks new ground by using Bogor's sophisticated support for OO language features to capture more directly the structure of real-world component and middleware systems, by tailoring a variety of aspects of model-checking algorithms to the relatively complex threading model of the ACE/TAO real-time event-channel, and by incorporating the verification framework into a robust development environment that is being used by researchers at several industrial sites including Boeing, Lockheed-Martin, and Rockwell-Collins to develop realistic systems.

The rest of this paper is organized as follows. Section 2 provides a brief overview of the characteristics of systems in the domain that we are considering, Section 3 describes how Bogor's modeling language is customized to more directly support systems in this domain, and how Cadena design notations are translated into Bogor, Section 4 describes how modules of Bogor's built-in checking engine are replaced with customized modules that implement distributed quasi-cyclic search, Section 5 provides experimental results of applying the framework, Section 6 discusses related work, and Section 7 concludes.

2 Component-based Avionics Systems in Cadena

We now give a brief overview of the structure of DRE systems that are designed using Cadena, and we explain how they give rise to quasi-cyclic state-spaces.

Figure 1 presents the CORBA component model (CCM) architecture for a very simple avionics system that shows steering cues on a pilot's navigational display. Although quite small, this system is indicative of the kind of system structure found in Boeing Bold Stroke designs. In the system, 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.

[2] Cadena project website: `http://cadena.projects.cis.ksu.edu`

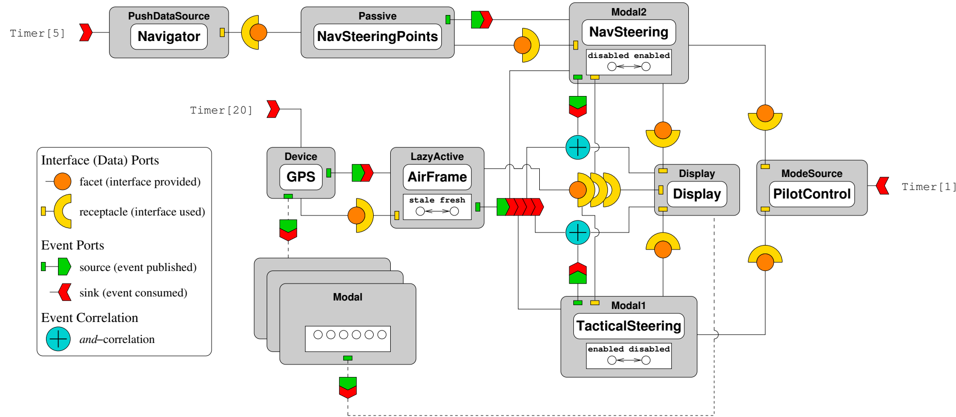


Fig. 1. Simple avionics system

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's 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. NavSteering and TacticalSteering are referred to as *modal components* since they each contain a *mode variable* (represented as a component attribute) whose value (enabled,disabled) determines the component's behavior. When a steering component is in enabled mode, it processes data from AirFrame and notifies the Display component that its data be is ready to be retrieved over one of the modal component's interface connections. When a steering component is in disabled mode, all incoming events from AirFrame are ignored and the component takes no other action.

There are many interesting aspects to this system and its development in Cadena that we cannot explain here due to lack of space (see [7] for more details). We focus here on issues related to the quasi-cyclic structure of the state-spaces of these systems.

In Bold Stroke applications, even though at a conceptual level component event source ports are connected to event sink ports, in the implementation, 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* (i.e., they do not contain threads) – instead, 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. The Event Channel also provides event correlation and event filtering mechanisms. In the example system of Figure 1,

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.

Periodic processing in Bold Stroke applications is achieved by having a component such as GPS subscribe to a periodic time-out (e. g. `Timer[20]`) that is published by the real-time event-channel (the event-channel contains dedicated timer threads to publish such events). The time between two occurrences 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).

In constructing transition system models of Bold Stroke applications, 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 r 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 1, the frame of the slowest rate (1 Hz) can be divided into 5 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*.

We do not keep an explicit representation of clock ticks, but instead our 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. Combining these constraints with the scheduling strategy implemented in Bogor for Cadena models safely approximates all interleavings of timeouts and component actions (given the assumption that no frame overruns occur) [7].

Systems such as the one in Figure 1 are captured in Cadena using specifications phrased in three parts: (1) component behavioral descriptions the describe the interface and transition semantics of component types such as the LazyActive component type in Figure 1 of which AirFrame is an instance, (2) a reusable model of a real-time CORBA event channel, and (3) system configuration information that describes the allocation of component instances and the port connections made between each of these instances as diagramed in Figure 1 (see [7] for a detailed explanation).

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.

```
CAD.Component EventChannel, GPS, AirFrame, NavDisplay, ... ;

EventChannel := CAD.createComponent();
GPS := CAD.createComponent("GPS");
AirFrame := CAD.createComponenet("AirFrame");
NavDisplay := CAD.createComponent("NavDisplay");

// create event-producing port
CAD.addSubscriberList(EventChannel, "timeOut20");
...
// connect EventChannel.timeOut20 to GPS.timeOut
{|tempComSub|} := new ComponentSubscriber;
{|tmepComSub|}.handlerFunction := EventHandlerType.{|common.BMDevice.timeOut<handler>()|};
{|tempComSub|}.portName := "timeOut";
{|tempComSub|}.component := GPS;
{|tempComSub|}.isSynchronous := false;
{|tempComSub|}.dispatchRate := 20;
CAD.addSubscriber<Subscriber>(EventChannel, "timeOut20", {|tempComSub|});
...
// create data-providing (interface) port
{|tempPort|} := CAD.createPort();
CAD.setPortMethodHandler<...>(
    {|tempPort|},
    "data<get>",
    {|common.ReadData.data<get>()|}.{|common.BMLazyActive.dataOut.data<get>()|});
CAD.registerPort(AirFrame, {|tempPort|}, "dataOut");

// connect consumer of data-providing port
CAD.connectPorts(Pair.create<...>(NavDisplay, "dataIn"), Pair.create<...>(AirFrame, "dataOut"));
```

Fig. 2. ModalSP system assembly using Bogor middleware primitives (excerpts)

3 From Cadena Scenarios to Bogor Transition Systems

3.1 Re-usable Middleware for Event Propagation

Our domain customization of Bogor begins with the introduction of primitive types for components, ports, events in BIR (Bandera Intermediate Representation) – the input language of Bogor. First a `CAD.Component` BIR object is created for each real-world component. Above this, a `CAD.Port` instance is associated with a host component for each *provided* interface port in the component's static description. The extension also lifts event publication and subscribership to first-class citizenship by defining a set of operations (e.g., addSubscriberList() and addSubscriber()) to allow simple maintenance of the event propagation chain. By making interconnections among the `CAD.Port` instances (for facets and receptacles) and subscribership records between event producers and sinks analogous to actual system deployment, we create an object-oriented BIR system which structurally mimics the real-world component software.

Figure 2 illustrates the sequencing operations on the middleware primitive ADT's to assemble a fragment of a component model software system. As is the case for all Bogor extensions, each of these high-level API operations is implemented by a Java method on the extension's class definition. Each new abstract datatype introduced into BIR is implemented by a Java class. In order for Bogor to correctly distinguish among instances of an ADT, the developer must explicitly choose a bit pattern representative of the object's state, to encode into the model checker state vector. We leverage this by encoding only a minimal set of information about the component: the values of any internal mode variables, external connection information, and subscriber list members. This allows an often dramatic reduction in the amount of storage required to maintain the seen-state set. By not encoding the minutia of a component ADT's implementation into the state vector, we also facilitate reductions by merging the state-vector representation of semantically identical but mechanically different data states.

The middleware datatype primitives for components, ports, and relatives are supplemented by BIR-language library routines (not shown) to simulate the action of an event channel in multiplexing event messages. During execution, a component c's method publishes a message from a registered event source port by invoking the fireEventFromComponent() function, passing the name of the outbound event port as an argument. This library function performs the generic

work of retrieving subscribership lists for the particular event source port on c, and then for each receiver of the event either directly invoking a handler function or queuing the message for later delivery if it crosses thread boundaries.

3.2 Direct use of OO Idioms for Data Communication

All data ports (facets and receptacles) conform to an *interface type*. The users of a component's data port are prohibited from making any assumptions about the implementation of said provided port. In effect, this makes all invocations of a port's operations *virtual method* calls. Bogor's native support for virtual method dispatch allows us to easily achieve the desired dispatch behavior. Virtual dispatch tables are declared using either an enumerated type or nodes in a type hierarchy as the lookup keys.

Because our Bogor implementation of components uses only one type, we instead create a special enumerated type for each method appearing on an interface type used in the scenario. The domain of each such enumerated type contains one element for each concrete component's implementation of the virtual method. A BIR virtual dispatch table is defined for each enumerated type, mapping the domain elements to the corresponding implementation of the virtual function. Supposing that the interface type `Data` includes a method `foo()`, and two component ports provide a `Data` port: `Sensor.dataOut` and `Proxy.dataOut`. Then the following enumerated type and virtual function table are created:

```
enum {|Data.foo()|} { {|Sensor.dataOut.foo()|}, {|Proxy.dataOut.foo()|} }

virtual {|Data.foo()|} on {|Data.foo()|} {
    /* enum value */              /* BIR function name */
    {|Sensor.dataOut.foo()|} -> {|Sensor.dataOut.foo()|}
    {|Proxy.dataOut.foo()|}  -> {|Proxy.dataOut.foo()|}
}
```

The observant reader will notice that we have used the same name for both an enumerated type identifier and a function identifier. In BIR, the namespaces of functions and enumerated types are separate. The identifier overloading here is deliberate; it makes a clear correlation between the mechanism of indirection (enumerated type values) and the function really being called by an invoke virtual BIR command.

3.3 Component Behaviors as Transition Systems

The heart of Cadena component specifications are the intra-method Cadena Property Specification (CPS) *transition system specifications*. As discussed in Section 2, Cadena models focus on expressing mode-conditioned execution control, port method invocations, and storage/retrieval of data values rather than complicated numerical computations that are often present in avionics applications.

Client Port Method Invocations: We have described in Section 3.2 an approach for invoking virtual methods on components. Calling port methods on remote components requires an extra step of indirection beyond this; the receiving component and the interface method's virtual dispatch table key must be determined first. This is accomplished by introducing a wrapper function for calling each different interface virtual method. The calling component's method body invokes the wrapper function, passing as arguments the client component instance, client port name, and desired method name. The wrapper function then consults the port interconnection information registered (as seen in Figure 2) to retrieve the remote component instance and function reference. A virtual method call is made on the provider component.

Variable Uses: Simple private variables and mode variables may both be used in CPS expressions and as arguments to port method calls. These variables have persistent values and are allocated in a per-component-instance basis. The basic Bogor middleware extension module defines polymorphic getAttribute() and setAttribute() operations on the `CAD.Component` datatype. CPS variables are stored and retrieved directly using this API; l-value occurrences of a variable are converted to setAttribute updates, and r-value uses are replaced inline with getAttribute expressions.

Conditionals: CPS allows both standard if branches and case switches on mode variables. These are both translated directly as BIR guarded commands. One must take care to avoid inadvertently blocking a transition by failing to add a fall-through case for a missing else branch or default case; this is easily accomplished by adding an extra guarded transition at the furcation point whose enabling condition is the conjunction of every other branch's test condition negated.

3.4 Custom Scheduler for Reducing Interleavings

As explored in [7], the target environment for Cadena system designs—a real-time processor with tightly controlled scheduling policies—allows dramatic reductions by preventing the exploration of impossible thread interleavings. Cadena component code is executed by threads from a priority-based pool. Further, external verification techniques can decide whether a scenario configuration satisfies timing requirements (that is, whether frame overruns occur). Accordingly, all thread interleavings in which a lower-priority event dispatcher preempts a higher-priority thread are irrelevant. Similarly, any interleaving which corresponds to the system timer delivering a fresh batch of jobs before the current jobs are finished, is infeasible.

A custom Bogor scheduling module leverages these domain insights. At a high level, Bogor's default scheduler module simply calculates the set of enabled transitions given the current model state. Our distributed real-time scheduling module first retrieves the default scheduler's calculated set of enabled transitions, then deletes any transitions which correspond to priority inversion or frame overrun. The state introspection facilities in Bogor allow a sophisticated analysis of each thread's stack variables to make such decisions about relative priorities of threads possible (we have chosen to make the event-dispatching thread code generic; it is instantiated many times per scenario configuration).

3.5 Extending Bogor to Inform Domain-specific Counterexamples

The modeling approach described here introduces several new abstract datatypes (e.g., `CAD.Component`, `Correlator.type`, `List.type`, etc.). While it is possible to use custom types in Bogor without providing counterexample detail, the resulting error traces are devoid of all state information for the domain-specific modules. Presumably, this is undesirable since the ADT is central to the analysis being undertaken.

To facilitate the debugging and analysis of counterexample witnesses, the middleware ADT modules each implement a Java method which writes a series of schema-governed elements into the overall counterexample file (itself an XML document). This grammar used to encode a custom datatype instance's state information allows unrestricted nesting of heap and primitive types inside an

extension type. This allows nested unfolding of any BIR objects "buried" inside a container ADT. Our hands-on experience shows this to valuable when debugging the implementation of Bogor modules themselves.

3.6 Automatic Creation of Bogor Transition Systems

Cadena contains a model generator that automatically translates CPS transition system into BIR. Architecturally, the model generator is a large VISITOR pattern implementation. By walking the structure of CPS syntax trees, the translator can produce appropriate BIR codes to simulate the dataflow behaviors (see Section 3.3) of each component method. These are packaged inside the BIR functions alluded to as virtual method call targets (Section 3.2). After creating the virtual method dispatch tables of Section 3.3, the generator then writes a system assembly phase (executed by the main thread before any others are forked) in which component instances and ports are allocated, inter-port connections are configured, and event dispatch queues are initialized as described in Section 3.1.

4 Distributed Quasi-cyclic Search

The time and storage costs required to explore the state space of a BIR transition system representative of a Cadena model increase, at a high level, with the number of permutations among all components' mode variables. Nondeterminism in the CPS dataflow statements is the primary source of branching in the transition systems, since the domain-specific scheduling policy makes most context switches deterministic. We now present an overview of a hybrid search algorithm which leverages these observations to make the state space exploration partitionable and, therefore, distributable among many processors.

Quasi-cyclic Search: A state transition system is defined to be an ordered quadruple $\langle S, s_0, E, \rightarrow \rangle$, where S is the set of all possible states, s_0 is the initial state, $E \subseteq S$ is the set of final states, and $\rightarrow \subseteq S \times S$ is the transition relation. $\rightarrow$ is neither required to be defined on all inputs nor is necessarily a function.

Each state in a transition system is defined to be one valuation of all the system's state variables V. Each state variable v is either an explicit variable (e.g., global data) or an implicit program counter variable.

A classical depth-first search (DFS) exploration of a transition system's *state space* (the set of all $s \in S$ reachable from the start state s_0 by a possibly empty sequence of applications of transitions in $\rightarrow$) accords no variables in a state's *vector* of values any special status. Indeed, the values of variables serve only to distinguish one state from another. By maintaining a global *seen state set*, the depth-first algorithm is able to detect cycles and prune the exploration of a state whose successors have already been visited.

The *quasi-cyclic search* (QCS) [9] is a strategy for coping with exploding memory requirements. It seeks to decompose the exploration of a transition system from one large global traversal into the traversal of many smaller, independent sub-state spaces. For example, event-driven systems feature an event-dispatching thread which blocks at a well-known control point waiting for work. At this point, a large chunk of its data structures (event queues, auxiliary lists) have predictable contents (likely: empty). Formally, we systematize this intuition by saying that a *transient* subset $T = \{t_1, t_2, \ldots, t_n\} \subseteq V$ of the system's state variables repeatedly takes on the distinguished values $\langle v_1, v_2, \ldots, v_n \rangle$.

In order to decompose an overall state space into independent regions, then, a QCS identifies those states for which T variables take on these distinguished

```
1   Procedure QCS()                          9   Procedure RegionDFS(s)
2     seen_g := ∅                            10    workSet := enabled(s)
3     RegionDFS(s_0)                         11    while workSet ≠ ∅
4     while ¬queueEmpty()                    12      α ← workSet.remove()
5       s_g := dequeue()                     13      s' := α(s)
6       seen_g := seen_g ∪ {s_g}             14      if p(s')
7       seen_t := {s}                        15        if s' ∉ seen_g ∧ ¬inQueue(s')
8       RegionDFS(s_g)                       16          enqueue(s')
                                             17      else
                                             18        if s' ∉ seen_t
                                             19          seen_t := seen_t ∪ {s'}
                                             20        pushStack(s')
                                             21        RegionDFS(s')
                                             22        popStack()
```

Fig. 3. Quasi-cyclic search algorithm

values. To do this, a boundary-state predicate $p : S \rightarrow \mathbb{B}$ is defined so that

$$p(s) \iff (t_1 = v_1 \wedge t_2 = v_2 \wedge \ldots \wedge t_n = v_n)$$

The state space exploration is accomplished by doing a hybrid DFS-BFS traversal. Beginning with the initial state s_0, a depth-first search proceeds as with classical state space exploration. The search is pruned whenever a state s satisfying $p(s)$ is reached. Rather than proceed (past such an s) down such paths, each such s is recorded in a pending-work queue. Each state space tree beginning from a seed state and bounded by either terminal states, internal repeated states, or p-satisfying states is called a *region*. When each region's DFS terminates, a p-satisfying state s_b is retrieved from the pending-work queue and another DFS is initiated from s_b. This algorithm is given in Figure 3.

Identification of Transient and Long-lived Variables: The size and placement of decomposed regions in a QCS state exploration depends entirely on the choice of p. The domain modeler must select conditions that accurately reflect a meta-"reset" condition. In a graphical user interface, a p which holds when there are no pending user input events and the main control loop is at its initial location is likely a good choice. For real-time systems with priority scheduling, the developer may select a p which is satisfied when all jobs are completed and the predictable delivery of work units is about to arrive (start-of-frame). Cadena systems are similar to both cases (both user input and frame boundaries are present). A quasi-cyclic search for Cadena uses a p which holds exactly when the following conditions occur:

- The system abstraction of time (which wraps after each hyperperiod) is at its initial value: 0;
- The thread designated to generate system timeouts is blocked (e.g., not currently creating a timeout); and
- All event queues are empty and each event-dispatching thread is blocked awaiting work.

Intuitively, these conditions correspond to the beginning of a new hyperperiod with no frame overrun. The long-lived values such as mode variables and event correlation automata are not tested by p (and are thus in the *non-transient* set) because they influence the inter-frame behavior. We remark at this point that no choice of p can result in an incomplete or incorrect state space exploration; the selection of a transient variables set T only affects the performance of QCS.

Adapting QCS to Distributed State Space Exploration: Because the region searches conducted by the RegionDFS procedure in Figure 3 do not rely

```
1   Process Coordinator()
2     ⟨ seen_g := ∅
3       tasks := 0
4       enqueue(s_0)
5     ⟩
6     ⟨await queueEmpty() ∧ tasks = 0⟩
7
8   Procedure GetSeed()
9     while true
10      ⟨await ¬queueEmpty() ∨ tasks = 0 →
11        if ¬queueEmpty()
12          s_temp := dequeue()
13          if s_temp ∉ seen_g
14            seen_g := seen_g ∪ s_temp
15            tasks := tasks + 1
16            return s_temp
17          else
18            continue
19        else
20          return nil
21      ⟩
22  Process RegionSearcher()
23    while true
24      s := GetSeed()
25      if s is nil
26        break
27      seen_t := {s}
28      RegionDFS(s)
29      ⟨tasks := tasks − 1⟩
30
31  Procedure RegionDFS(s)
32    workSet := enabled(s)
33    while workSet ≠ ∅
34      α ← workSet.remove()
35      s' := α(s)
36      if p(s')
37        <enqueue(s')>
38      else
39        if s' ∉ seen_t
40          seen_t := seen_t ∪ {s'}
41        pushStack(s')
42        RegionDFS(s')
43        popStack()
```

Fig. 4. Distributed quasi-cyclic search algorithm

on any external data (except the global seen-before boundary state set), the algorithm is amenable to parallelization. This is done, in broad terms, by making the old RegionDFS into an active process. Figure 4 gives an adaptation of standard QCS to a distributed environment. We have used angle brackets ("⟨" and "⟩") to denote critical section code running in mutual exclusion, in the manner of SyncGen specifications [8]. A distinguished instance of the Coordinator process maintains the global seen-before boundary state set $seen_g$, the counter of active tasks ($tasks$), and pending-work queue. As many RegionSearcher processes as desired are initiated to consume boundary states from $seen_g$ and explore the decomposed state space one region at a time.

Architecture: We have adapted Bogor to the distributed version of QCS by writing a new state graph traversal module and substituting it for the standard searcher in the RegionSearcher client process. Each of the subordinate client processes requests a unit of work (by invoking the equivalent of what we have shown as GetSeed), which is a tuple $< b, c, s >$ where b is the BIR transition from which the region to be checked is excerpted, c is the Bogor system configuration (runtime options, etc.), and s is the seed state (the formal parameter s of the RegionDFS procedure). The subordinate process constructs a complete Bogor system as directed by c, uses the lexical frontend to load the BIR system b, and finally starts a model check from state s which is bounded by the predicate p (this is encoded inside c). As the model check proceeds, each p-satisfying state encountered is reported back to the Coordinator process.

Transporting State Information: Bogor is implemented in Java. Fortunately, the Remote Method Invocation (RMI) subsystem bundled directly supported in Java provides an effective mechanism for copying state variable data to and from the Coordinator process. A deep clone of the Bogor object containing all state information is automatically serialized and reconstructed on the remote host during an RMI call. All that remains to acclimate the state object to its new Bogor host process. An enhancement to the main Bogor state API allows the state object to re-acquire references to any required runtime modules.

Facilitating Counterexamples: Constructing counterexamples from a standard DFS is simple: a simple examination of the backtracking information stack reveals the complete path to a violating state. In QCS, there is no "current" path from the root state to a property violation. The $seen_g$ set contains a series of p-states whose successors have been explored. Our implementation supplements

this with two auxiliary lookup tables: (1) a map between each p-satisfying state and parent seed state, and (2) a map from each p-satisfying state and the series of backtracking steps which lead to the parent seed state. When a violating state is reached, all backtracking path fragments are concatenated to form a complete trace.

5 Results

To evaluate the scalability of our parallel quasi-cyclic search, we have chosen an updated version of a suitable system from the original experiments run on quasi-cyclic systems [9].

The specific model we chose to test is the result of automatically compiling the Cadena input artifacts for a ModalSP scenario *plus* two additional modal components, into representative BIR code. In the notation of the scenario configurations used in [9], this BIR system would be called a $\langle 2, 2, 2\rangle$ configuration.

The experimental platform consisted of 4-processor AMD Opteron 842 systems running at 1.6 GHz with 16 GB of memory each, running SuSE Linux 8.2 (beta release for AMD64). The nodes are connected with a 1000 Mbps copper ethernet switch. Bogor ran on top of the 1.5.0-beta-b32c native AMD64 Java Virtual Machine from Sun. Client processes were allocated 8 GB maximum heap size (half the total physical memory of its host).

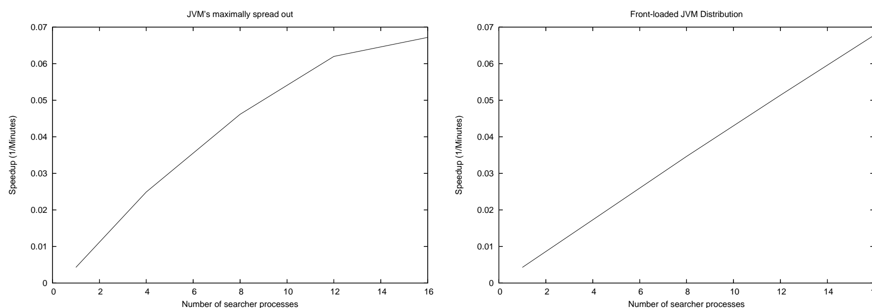
Five total machines were used; in every instance one was reserved exclusively for the coordinating server; the remaining four were used to execute client processes.

Our initial series of trials was done by running $\frac{n}{4}$ out of n total client processes on each machine. The sequential QCS of the model took Notice that the performance gains drop off steeply after 8 client processes are allocated in series (a) of Figure 5. This was something of a mystery to us, because the CPU and I/O load on the coordinator server never reached high levels. Why could our system only weakly support additional parallelism?

After careful examination, we noticed that the turning point in the performance curve on (a) corresponded to the point where the total size of all JVM heaps allocated on the client processes equal the amount of physical RAM available on the host. That is, the JVM's were consuming $\frac{8}{4} \times 8$ GB= 16 GB per host at this point. Increasing the total number of client processes past 8 implied double-booking the physical RAM on the hosts. Although the black-box nature of the Java Virtual Machine's memory manager prevents a detailed analysis, we surmise that memory contention prevented more than 2 client processes from fully utilizing the processor time on any one machine.

To test this hypothesis, we re-ran our test series. As before, additional client processes were added in groups of 4. This time, though, each new 4-processes client group was embodied as a fully loaded machine. This series, (b) in Figure 5, shows the performance gains of adding one *fully utilized* client machine at a time. The speedup curve is virtually linear.

Despite some problems getting full utilization of each client machine when very high RAM allocations are desired, our architecture for distributed quasi-cyclic searching appears to be intrinsically quite scalable. This should continue to be the case so long as (1) the cost of copying states is dominated by the time to search an average region and (2) the models themselves have sufficient nondeterminism to usually keep at least as many states in the unexplored-boundary-state set as there are client processors.



(a) Adding each new client to least-loaded machine

(b) Adding one fully-loaded client machine at a time

Fig. 5. Scalability graphs

We have additionally run many smaller models (also generated by Cadena) through the distributed QCS system (BasicSP, standard ModalSP, MediumSP, etc.). While less systematic than the data presented here and less persuasive because their running times are much lower, these transition systems exhibited roughly the same performance improvement curves as additional client processes were brought online.

6 Related Work

Garavel et al. present a technique for distributing an explicit state space exploration across multiple network computers in [10]. By carefully choosing a hash function to use for partitioning states uniformly across network nodes, and by implementing non-blocking Send and Recv operations between model-checking processes, Garavel et al. improved the scalability of a SCSI subsystem model to near-linear [10]. Our work on quasicyclic search does not use a static partitioning; rather a central pool of pending seed states is maintained and each node retrieves one as it becomes free. Additionally, we can afford to use synchronous analogues of the Send and Recv operations because intra-region searches typically execute very long sequences of transitions before interrupting to perform network operations. We do not believe network messages impose a noticeable bottleneck on sytem performance.

The Java PathFinder (JPF) model checker [17] uses a dynamic partitioning scheme in which the state-to-host hashing function changes when analyses indicate that the ration of network messages to transition execution could be improved by redefining the state partitioning scheme. Significant gains are made versus the baseline JPF static partitioning method. As mentioned in context of [10] above, the quasicyclic search instead reduces inter-node network traffic by decomposing a state space into many *independent* regions; the entirely of each is explored by one model checker process without network traffic interruptions.

Jones and Mercer improve on the static partitioning typically used in distributed model checking by *randomly* choosing the next state to expand from a set of priority-sorted frontier states [16]. In some cases where a Bayes heurestic search algorithm required many transitions to reach a first error state, introducing randomness into the search order significantly reduced the number of transitions executed before reaching a first error state. Our quasicyclic search in general seeks to rely on domain knowledge to improve parallel search performance (Jones and Mercer specifically attempt to develop model-independent

methods), but in principle the approach of [16] could be used to inform intra-region searches.

Ben-David et al. in [1] and later Heyman et al. in [13] give algorithms for achieving load-balancing across the network nodes during model checking. The former achieved counterexample generation without requiring any one network node's process to contain a complete state set and significantly reduced over-all memory requirements. The latter builds on this approach by using adaptive partitioning of the state space and compact Binary Decision Diagrams (BDD's) which can be transferred over a network, to further decrease memory requirements. Because these approaches are adaptations of the symbolic model checking algorithm, it is an open question whether they can be applied to our setting; the Bogor framework uses an explicit, domain-tailored state-space exploration algorithm.

7 Conclusions

By leveraging various properties of the Bold Stroke avionics domain via customization of Bogor, we have been able to reduce the cost of model checking by multiple orders of magnitude compared to our earlier attempts [12] to model these systems in dSpin. Multiple research projects besides our own have been able to customize Bogor effectively to particular domains. Still, even though we believe Bogor is much easier to customize than many existing model checkers, a fair amount of knowledge of the internals of a model checker is sometimes required for such efforts. The Bogor website provides a variety of educational, tutorial, and example materials to ease the burden of learning the infrastructure, and we continue to search for additional automated techniques to aid developers in building, testing, and debugging their own Bogor extensions.

We have been able to introduce model checking concepts to several different industrial research groups at Boeing, Lockheed-Martin, and Rockwell-Collins due to the incorporation of Bogor into Cadena. However, to date, engineers at these sites tend to make much more use of the design and structural analysis capabilities of Cadena rather than the model checking technology because they often do not want to put forth the effort required for writing transition models of components. To address this issue, we are exploring (a) techniques for inferring component transition models from run-time trace data and (b) opportunities for further leveraging component transition system models in other forms of analysis and code synthesis.

Acknowledgements: The authors would like to thank the members of the Santos group at Kansas State University for many useful discussions about issues related to this paper, especially, Jesse Greenwald, Georg Jung, and Xianghua Deng. The research reported in this paper 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 NSF (CCR-0306607) by Lockheed Martin, by Rockwell-Collins, and by an IBM Corporation Eclipse Award.

References

1. S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster. Scalable distributed on-the-fly symbolic model checking. In *Proceedings of Formal Methods in Computer-Aided Design*, pages 390–404, 2000.

2. D. Bosnacki, D. Dams, and L. Holenderski. Symmetric SPIN. *International Journal on Software Tools for Technology Transfer*, 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. W. Chan, R. J. Anderson, P. Beame, D. H. Jones, D. Notkin, and W. E. Warner. Optimizing symbolic model checking for statecharts. *IEEE Transactions on Software Engineering*, 27(2):170–190, February 2001.
5. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
6. C. Demartini, R. Iosif, and R. Sisto. dspin : A dynamic extension of SPIN. In *Proc. of the 6th International SPIN Workshop*, volume 1680, pages 261–276. Springer-Verlag, Sept. 1999.
7. W. Deng, M. B. Dwyer, J. Hatcliff, G. Jung, Robby, and G. Singh. Model-checking middleware-based event-driven real-time embedded software. In *Proceedings of the First International Symposium on Formal Methods for Components and Objects (FMCO 2002)*, November 2002.
8. X. Deng, M. B. Dwyer, J. Hatcliff, and M. Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *Proceedings of the 24th International Conference on Software Engineering*. IEEE Press, 2002.
9. M. B. Dwyer, Robby, X. Deng, and J. Hatcliff. Space reductions for model checking quasi-cyclic systems. In *Proceedings of the Third International Conference on Embedded Software*, 2003.
10. H. Garavel, R. Mateescu, and I. Smarandache. Parallel state space construction for model-checking. In *Proceedings of Eighth International SPIN Workshop*, volume 2057 of *Lecture Notes in Computer Science*, pages 217+. Springer-Verlag, 2001.
11. P. Godefroid. Model-checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 174–186, Jan. 1997.
12. J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. P. Ranganath. Cadena: An integrated development, analysis, and verification environment for component-based systems. In *Proceedings of the 25th International Conference on Software Engineering*. IEEE Press, May 2003.
13. T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In O. Grumberg, editor, *Computer-Aided Verification, 12th International Conference*, volume 1855, pages 20–35. Springer, 2000.
14. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.
15. 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.
16. M. Jones and E. Mercer. Explicit state model checking with hopper. In *Proceedings of Eleventh International SPIN Workshop*, volume 2989 of *Lecture Notes in Computer Science*, pages 146–150. Springer-Verlag, April 2004.
17. F. Lerda and W. Visser. Addressing dynamic issues of program model checking. In *Proceedings of Eighth International SPIN Workshop*, volume 2057 of *Lecture Notes in Computer Science*, pages 80–102. Springer-Verlag, 2001.
18. Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *Proceedings of the 9th European Software Engineering Conference held jointly with the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2003.