

Software Model Checking Using Bogor - a Modular and Extensible Model Checking Framework

3rd Estonian Summer School in
Computer and System Science (ESSCaSS'04)

Slide Set 08: Cadena Overview

<http://bogor.projects.cis.ksu.edu>
<http://www.cis.ksu.edu/~hatcliff/ESSCaSS04>

John Hatcliff

Matthew B. Dwyer

Robby

SAnToS Laboratory, Kansas State University, USA

Support

US Army Research Office (ARO)
US National Science Foundation (NSF)
US Department of Defense
Advanced Research Projects Agency (DARPA)

Boeing
Honeywell Technology Center
IBM
Intel

Lockheed Martin
NASA Langley
Rockwell-Collins ATC
Sun Microsystems

Acknowledgements

- Work on Cadena has been carried out by the following team of people
 - PIs: John Hatcliff, Matt Dwyer, Gurdip Singh
 - Primary Developers: Jesse Greenwald, Venkatesh Ranganath, Adam Childs, Prashant Kumar Shanti
 - Students: Georg Jung, William Deng, Matt Hoosier

Goals of the Cadena Project



An Integrated Development Environment for
Analysis, Synthesis, and Verification of
Component-based Systems

I. Platform for real-world experimentation with technologies for building high-assurance distributed systems using CORBA Component Model

... robust tool environment suitable for industrial experimentation
... model-based development, middleware configuration, and code synthesis

... light-weight specification, analysis, and verification techniques
... customizable to different domains/product lines

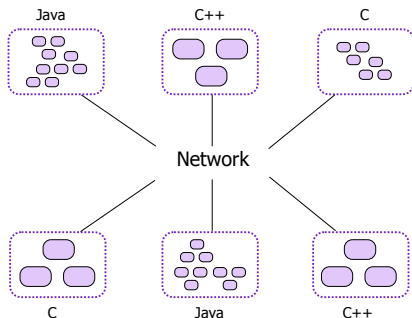
II. Avenue for collaborating with industrial research teams and middleware experts to guide next-generation component/middleware technology

... interacting with groups at Boeing, Rockwell-Collins, Lockheed-Martin to develop techniques that match fit into development process
... collaborating with middleware experts (e.g., ACE/TAO RT-middleware) to make frameworks more amenable to model-based configuration and analysis

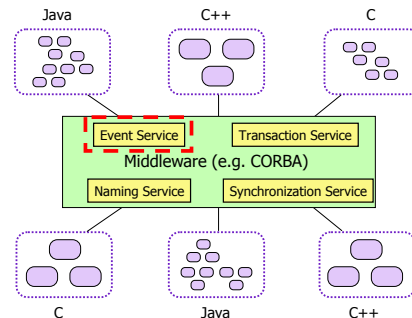
Lecture Outline

- Motivation for Middleware and Components
- Broad themes of Cadena
- A real-world test-bed from the avionics domain
- Main features of Cadena
 - component development
 - lightweight semantic annotations
 - intra-component dependences
 - intra-component transition semantics
 - system assembly
 - Analysis, automated design device, analysis driven configuration and customization of middleware and services
 - Extending Bogor's modeling language to support Cadena designs
 - Customizing Bogor's scheduling and state-space search modules to Cadena/BoldStroke designs

Distributed Components



Distributed Components



Objects To Components

- Consider: group of objects working together to provide a service to clients
- Objects are meant to be used "as a team"
 - unit of composition
- No language mechanism to identify components as a single group
 - explicitly define interfaces
 - explicitly define dependences on other 'groups'
- Harder for 3rd parties to reuse and assemble

Objects To Components

- Components collect related classes together to form a coarser-grain composable unit
- Components explicitly define interfaces they provide to their clients
- Components indicate the other interfaces/events they depend on
- Considerable auto-coding functionality provide

Checking CCM Systems

Modern Software Systems

Issues

- These systems are huge!
- Extensive use of OO patterns & software layering
- What are appropriate abstractions for formal reasoning?
- How can we help developers write them?
- Useful properties?
- How must conventional model-checking engines be extended?

Component-based Design

Component Development

Cadena development environment allows model-based development of Bold Stroke applications using the CORBA Component Model (CCM)

Component Development

- Development of component interfaces using CCM *Interface Definition Language*
- Automatic generation of component infrastructure code using CCM IDL compilers
- Development of core functional code (business logic) using Eclipse Java facilities

Leverage CORBA IDL

```

component BMLazyActive {
  provides ReadData outData;
  uses ReadData inData;
  publishes DataAvailable outDataAvailable;
  consumes DataAvailable inDataAvailable;
  attribute LazyActiveMode dataStatus;
};
  
```

IDL Compiler

Component Implementation Stubs & Skeletons

+

```

dependency default t
  == none;
dependencies {
  inDataAvailable | labi e
  -> outDataAvailable | labi e;
}
  
```

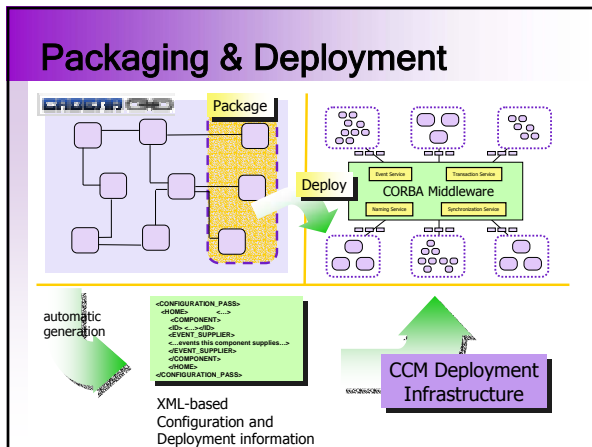
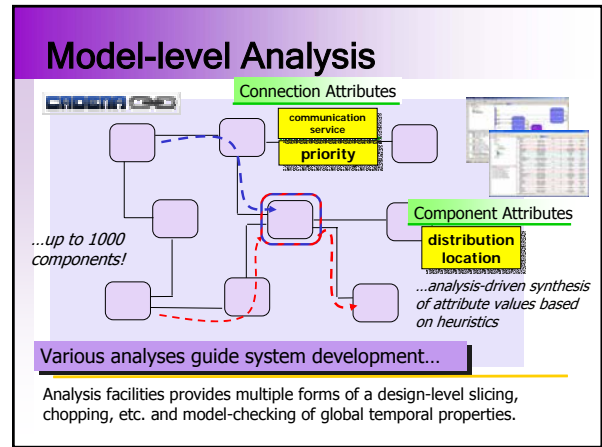
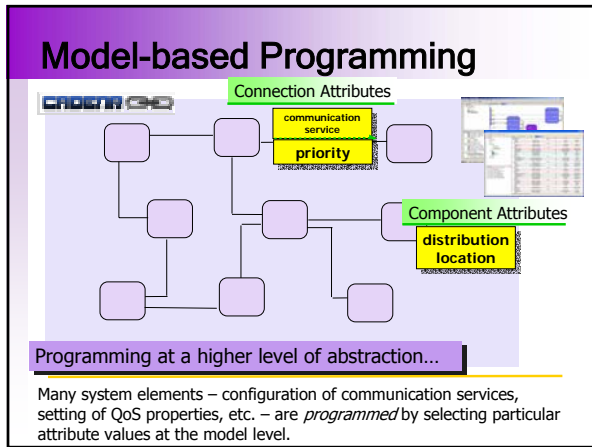
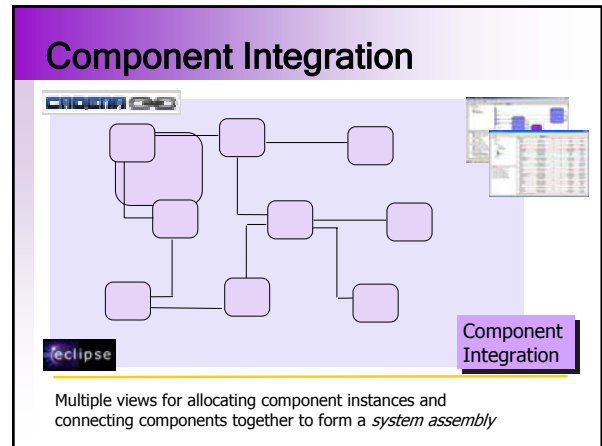
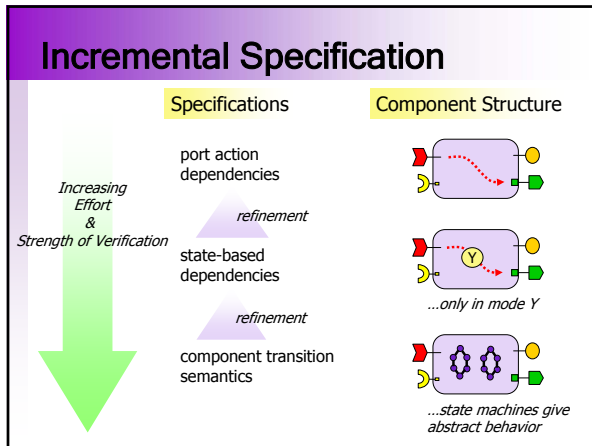
```

behavior {
  IF (mode==enabled) {
    push outDataAvailable | labi e;
  }
}
  
```

Dependency Annotations Transition System Semantics


Model Builder

Dependency Analysis and Model-checking Engine



- ## Lecture Outline
- Motivation for Middleware and Components
 - Broad themes of Cadena
 - A real-world test-bed from the avionics domain
 - Main features of Cadena
 - component development
 - lightweight semantic annotations
 - intra-component dependencies
 - intra-component transition semantics
 - system assembly
 - Analysis, automated design device, analysis driven configuration and customization of middleware and services
 - Extending Bogor's modeling language to support Cadena designs
 - Customizing Bogor's scheduling and state-space search modules to Cadena/BoldStroke designs

Avionics Mission Control Systems



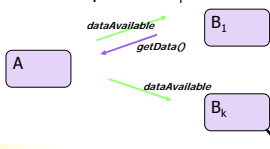
- Mission-control software for Boeing military aircraft
- Boeing's Bold Stroke Avionics Middleware
 - ...built on top of ACE/TAO RT CORBA

- Provided with an Open Experimental Platform (OEP) from Boeing
 - a sanitized version of the real system
 - 100,000+ lines of C++ code (including RT CORBA middleware)
 - 50+ page document that outline development process and describe challenge problems
- Must provide...
 - tool-based solutions that can be applied by Boeing research team to realistic systems
 - solutions that fit within current development process, code base, etc.
 - metrics for that allow Boeing research team to evaluate tool performance and ease of use

Control-Push Data-Pull

Typical situation

Component A computes some data that is to be read by one or more components B_i



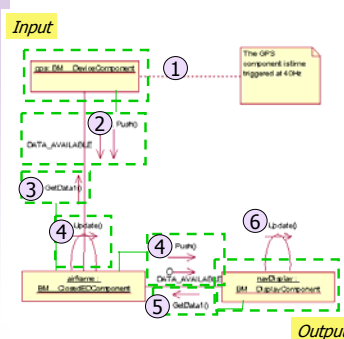
Run-time Actions

- A publishes a dataAvailable event
- B_i call the getData() method of A to fetch the data

Depending on current state, component may not fetch data

Control-Push Data-Pull Structure

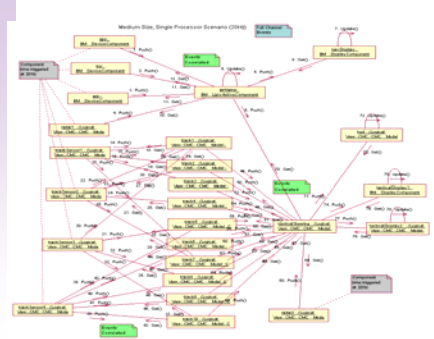
Input



1. Logical GPS component receives a periodic event indicating that it should read the physical GPS device.
2. Logical GPS publishes DATA_AVAILABLE event
3. Airframe component fetches GPS data by calling GPS GetData method
4. Airframe updates its position data and publishes DATA_AVAILABLE event
5. NavDisplay component fetches AirFrame data by calling AirFrame GetData method
6. NavDisplay updates the physical display

Output

Larger Configuration



...moving up to 1000+ components

System Requirements

Very idealized(!), but should give you the flavor

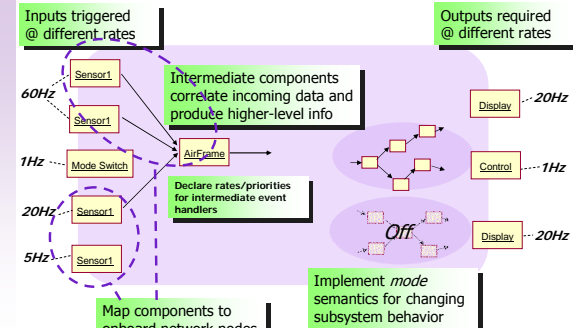
Input Requirements

- The system shall request new inputs from the GPS subsystem at a 40 Hz rate.
- The system shall poll for a pilot steering mode input at a 1 Hz rate.
- The system shall receive data from the navigator controls at a 5 Hz rate.

Output Requirements

- The system shall disable the display of steering information when deselected by the pilot.
- When the navigation steering mode is selected, the system shall:
 - Update navigation steering information display outputs at 20Hz rate based on current airframe data and the current list of navigation points that have been submitted by the navigator. The latency between the GPS data inputs and the display output shall be less than a single 20 Hz frame. The latency between navigation point input and the associated output shall be less than a single 5 Hz frame.
- When the tactical steering mode is selected, the system shall:
 - Update tactical steering information display outputs whenever the airframe position data changes.
- The system shall display new aircraft position data at a 20 Hz rate. The latency between associated inputs and this output shall be less than a single 20 Hz frame.

System Design Aspects



Inputs triggered @ different rates

- 60Hz: Sensor1
- 1Hz: Mode_Switch
- 20Hz: Sensor1
- 5Hz: Sensor1

Outputs required @ different rates

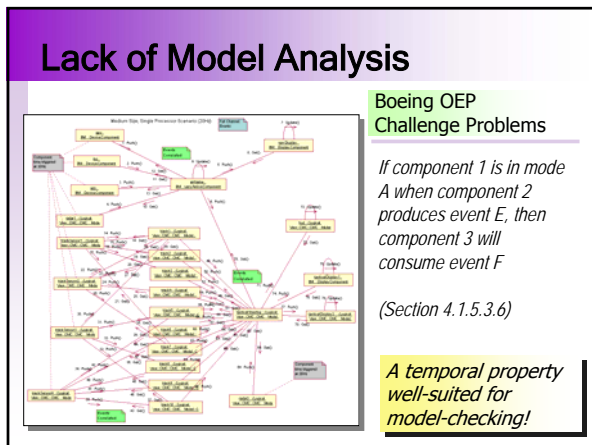
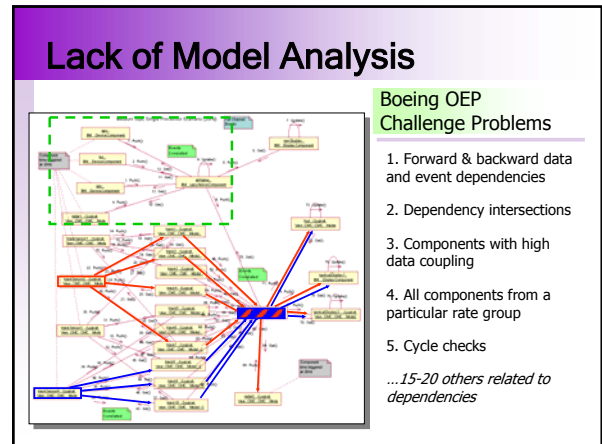
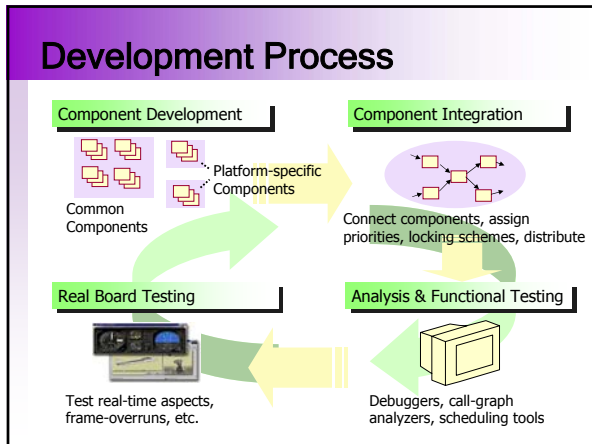
- 20Hz: Display
- 1Hz: Control
- 20Hz: Display

Intermediate components correlate incoming data and produce higher-level info

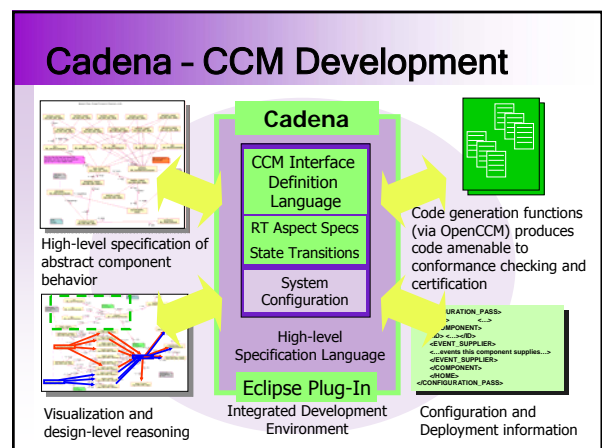
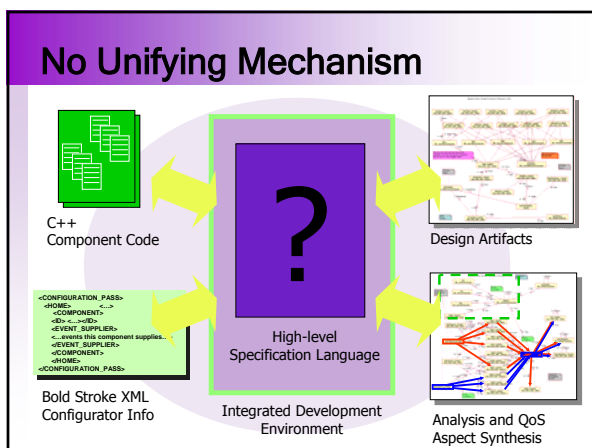
Declare rates/priorities for intermediate event handlers

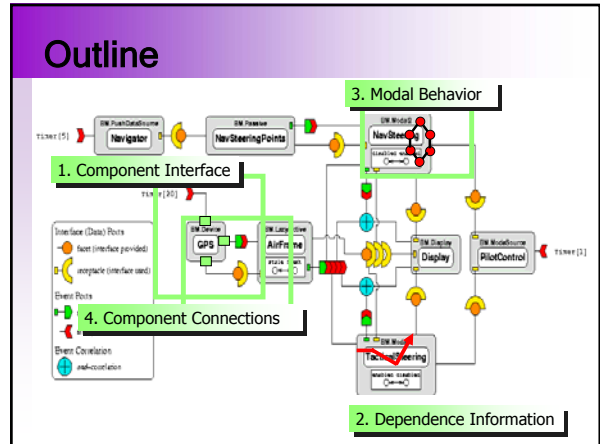
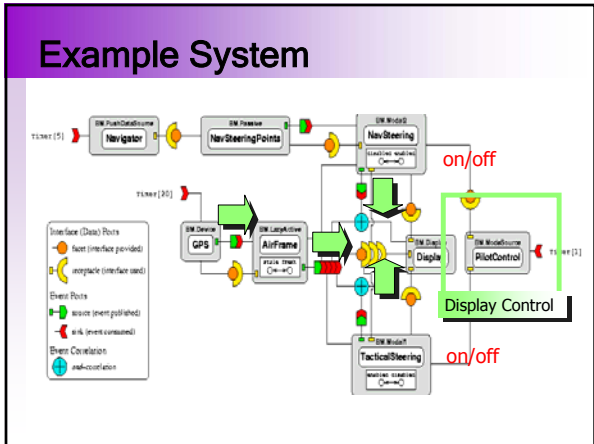
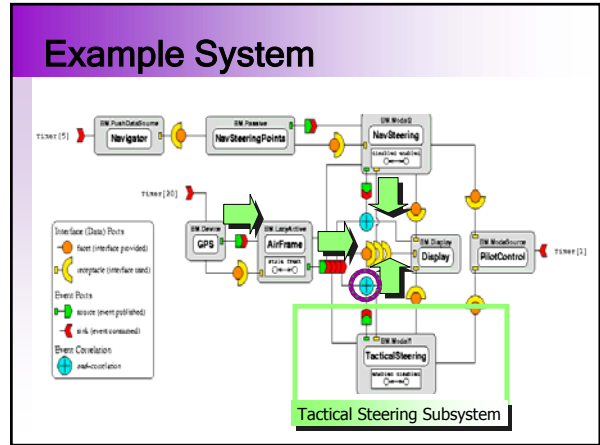
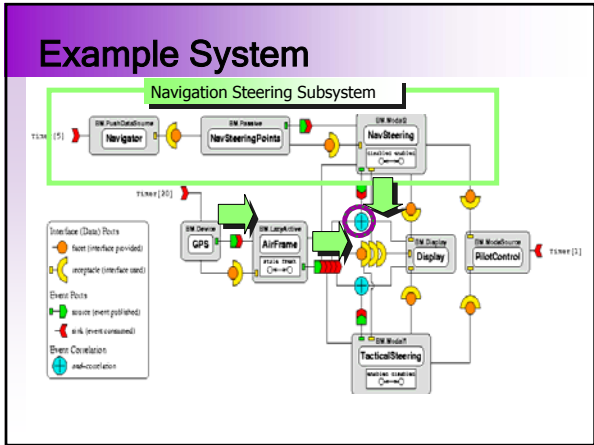
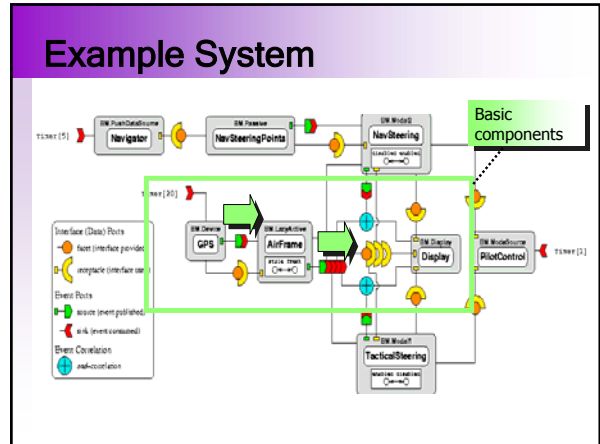
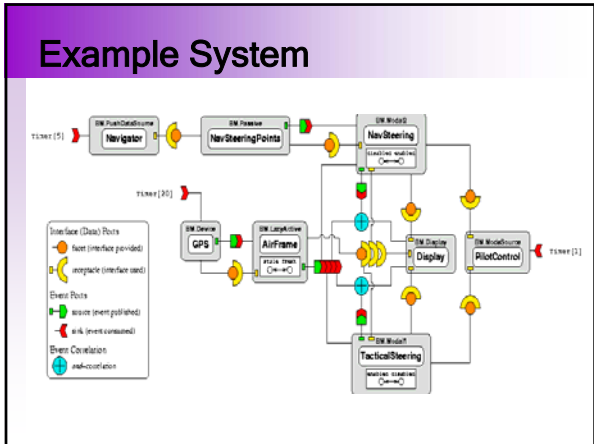
Map components to onboard network nodes

Implement mode semantics for changing subsystem behavior



- ## Lecture Outline
- Motivation for Middleware and Components
 - Broad themes of Cadena
 - A real-world test-bed from the avionics domain
 - Main features of Cadena
 - component development
 - lightweight semantic annotations
 - intra-component dependences
 - intra-component transition semantics
 - system assembly
 - Analysis, automated design device, analysis driven configuration and customization of middleware and services
 - Extending Bogor's modeling language to support Cadena designs
 - Customizing Bogor's scheduling and state-space search modules to Cadena/BoldStroke designs





Component IDL

```
component BMLazyActive {  
  provides ReadData outData;  
  uses ReadData inData;  
  publishes DataAvailable outDataAvailable;  
  consumes DataAvailable inDataAvailable;  
  attribute LazyActiveMode dataStatus;  
};
```

CORBA 3
CCM IDL
ModalSP Components

Component IDL

component BMLazyActive {
 provides ReadData outData;
 uses ReadData inData;
 publishes DataAvailable outDataAvailable;
 consumes DataAvailable inDataAvailable;
 attribute LazyActiveMode dataStatus;
};

CORBA 3
CCM IDL
ModalSP Components

Component IDL

component BMLazyActive {
 provides ReadData outData;
 uses ReadData inData;
 publishes DataAvailable outDataAvailable;
 consumes DataAvailable inDataAvailable;
 attribute LazyActiveMode dataStatus;
};

output data port
(facet)

CORBA 3
CCM IDL
ModalSP Components

Component IDL

component BMLazyActive {
 provides ReadData outData;
 uses ReadData inData;
 publishes DataAvailable outDataAvailable;
 consumes DataAvailable inDataAvailable;
 attribute LazyActiveMode dataStatus;
};

input data port
(receptacle)

CORBA 3
CCM IDL
ModalSP Components

Component IDL

component BMLazyActive {
 provides ReadData outData;
 uses ReadData inData;
 publishes DataAvailable outDataAvailable;
 consumes DataAvailable inDataAvailable;
 attribute LazyActiveMode dataStatus;
};

output event port
(event source)

CORBA 3
CCM IDL
ModalSP Components

Component IDL

component BMLazyActive {
 provides ReadData outData;
 uses ReadData inData;
 publishes DataAvailable outDataAvailable;
 consumes DataAvailable inDataAvailable;
 attribute LazyActiveMode dataStatus;
};

input event port
(event sink)

CORBA 3
CCM IDL
ModalSP Components

Component IDL

```

component BMLazyActive {
  provides ReadData outData;
  uses ReadData inData;
  publishes DataAvailable outDataAvailable;
  consumes DataAvailable inDataAvailable;
  attribute LazyActiveMode dataStatus;
};
  
```

mode attribute

CORBA 3
CCM IDL
ModalSP Components

Leverage CORBA IDL

```

component BMLazyActive {
  provides ReadData outData;
  uses ReadData inData;
  publishes DataAvailable outDataAvailable;
  consumes DataAvailable inDataAvailable;
  attribute LazyActiveMode dataStatus;
};
  
```

IDL Compiler

Component Stub & Skeleton Code

+

```

dependencydefault t == none;
dependencies {
  inDataAvailable |abl e
  -> outDataAvailable |abl e;
}
  
```

```

behavior {
  IF (mode==enabled) {
    push outDataAvailable |abl e;
  }
}
  
```

Model Builder

Dependency Annotations

Transition System Semantics

Dependency Analysis and Model-checking Engine

Incremental Specification

Increasing Effort & Strength of Verification

Specifications

- port action dependencies
- state-based dependencies
- component transition semantics

refinement

refinement

Component Structure

...only in mode Y

...state machines give abstract behavior

Outline

2. Dependence Information

Light-weight Dependency Specs

```

dependencydefault t == none;
dependencies {
  dataWriterOut.set_data() -> outDataAvailable;
}
behavior { ... }
  
```

call on set_data()

triggers

outDataAvailable port action

Light-weight Dependency Specs

triggers no other actions

```

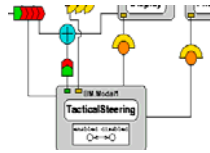
dependencydefault t == all;
dependencies {
  modeChange() -> {
    case modeChange.modeVar of {
      enabled: inDataAvailable;
      -> dataIn.get_data(),
      outDataAvailable;
      disabled: inDataAvailable ->
    }
  }
}
behavior { ... }
  
```

Light-weight Dependency Specs

```
dependency default == all;
```

```
dependencies {
  modeChange() ->;
  case modeChange.modeVar of {
    enabled: inDataAvailable
      -> dataIn.getData(),
      outDataAvailable;
    disabled: inDataAvailable ->;
  }
}
```

```
behavior { ... }
```



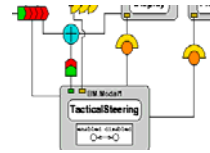
in enabled mode, shows actions triggered by receipt of event on inDataAvailable port

Light-weight Dependency Specs

```
dependency default == all;
```

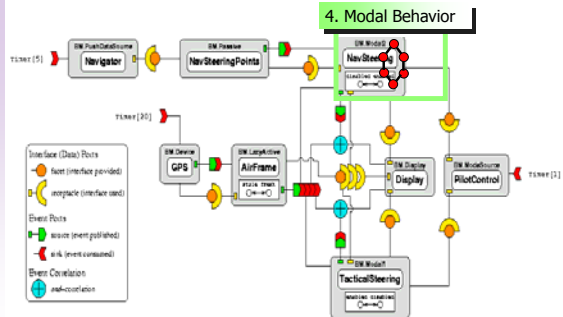
```
dependencies {
  modeChange() ->;
  case modeChange.modeVar of {
    enabled: inDataAvailable
      -> dataIn.getData(),
      outDataAvailable;
    disabled: inDataAvailable ->;
  }
}
```

```
behavior { ... }
```



in disabled mode, inDataAvailable triggers no other port actions

Outline



4. Modal Behavior

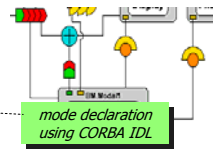
Component Behavior

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

```
enum Modes (enabled, disabled);
```

```
Modes m;
```

```
behavior {
  handles dataInReady (DataAvailable e) {
    case m of {
      enabled {
        dataOut::data <- dataIn.getData();
        push {} dataOutReady;
      }
      disabled {}
    }
  }
}
```



mode declaration using CORBA IDL

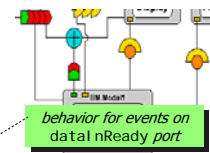
Component Behavior

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

```
enum Modes (enabled, disabled);
```

```
Modes m;
```

```
behavior {
  handles dataInReady (DataAvailable e) {
    case m of {
      enabled {
        dataOut::data <- dataIn.getData();
        push {} dataOutReady;
      }
      disabled {}
    }
  }
}
```



behavior for events on dataInReady port

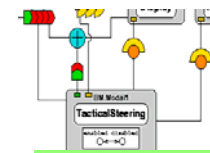
Component Behavior

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

```
enum Modes (enabled, disabled);
```

```
Modes m;
```

```
behavior {
  handles dataInReady (DataAvailable e) {
    case m of {
      enabled {
        dataOut::data <- dataIn.getData();
        push {} dataOutReady;
      }
      disabled {}
    }
  }
}
```



behavior mode cases

Component Behavior

```

component BModal {
  uses      ReadData dataIn;
  consumes DataAvailabl e InDataAvailabl e;
  publishes DataAvailabl e outDataAvailabl e;
  provides ReadData dataOut;
  provides ChangeMode modeChange;

```

```

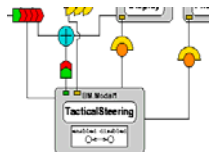
enum Modes (enabl ed, dl sabl ed);
Modes m;

```

```

behavi or {
  handl es dataInReady (DataAvailabl e) {
    case m of
      enabl ed {
        dataOut::data <- dataIn.g etData();
        push () dataOutReady;
      }
      dl sabl ed {}
    }
  }
}

```



data flow specification

Component Behavior

```

component BModal {
  uses      ReadData dataIn;
  consumes DataAvailabl e InDataAvailabl e;
  publishes ReadData dataOut;
  provides ChangeMode modeChange;

```

```

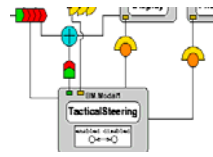
enum Modes (enabl ed, dl sabl ed);
Modes m;

```

```

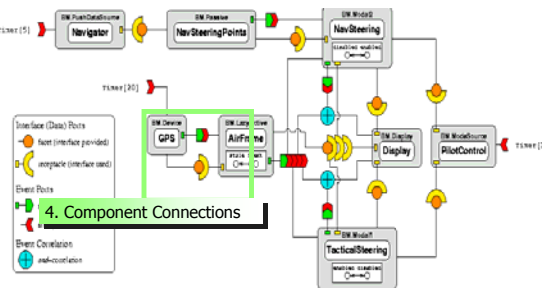
behavi or {
  handl es dataInReady (DataAvailabl e) {
    case m of
      enabl ed {
        dataOut::data <- dataIn.g etData();
        push () dataOutReady;
      }
      dl sabl ed {}
    }
  }
}

```



publish event

Outline



4. Component Connections

Three Synchronized Views

Scenario Description

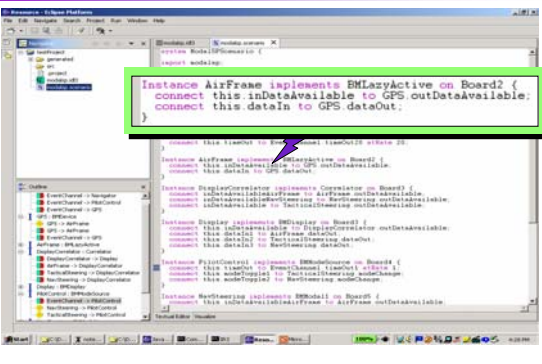
Graphical View

Spreadsheet View

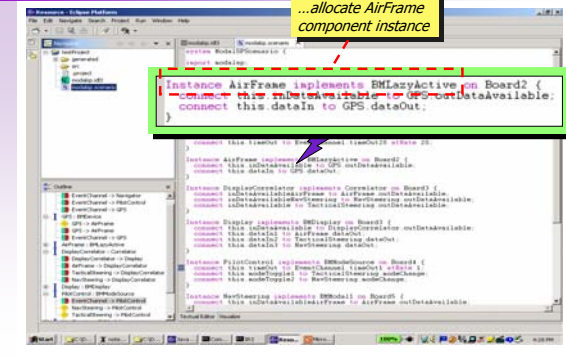
Textual View

Single Internal Representation

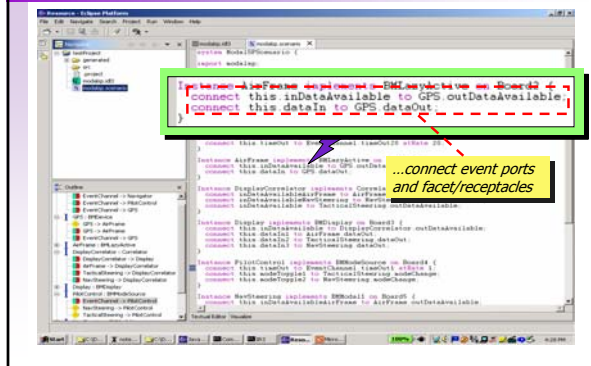
Textual View



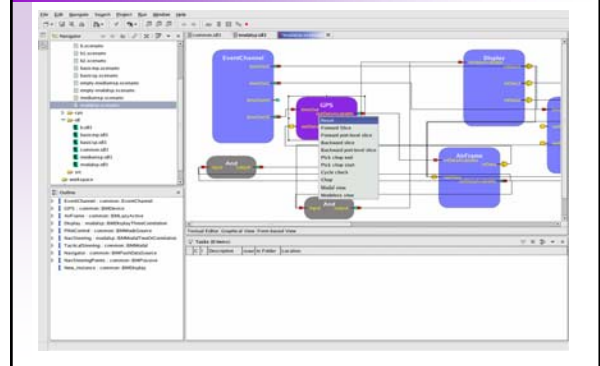
Textual View



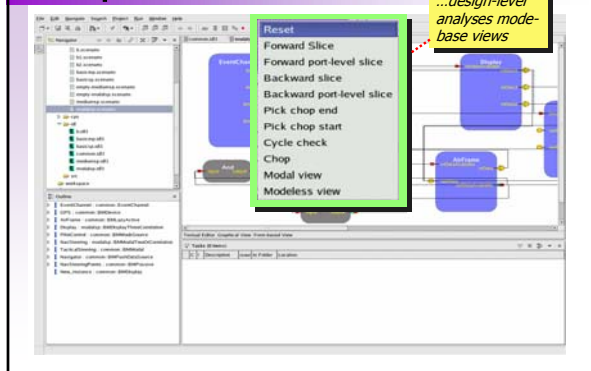
Textual View



Graphical View



Graphical View



Spreadsheet View

Component	Port Name	Value	Direction	Target Component	Target Port
GPS	outDataAvailable	20	<	Board1	inDataAvailable
Board1	dataIn	0	>	GPS	dataOut

Spreadsheet View

Spreadsheet View

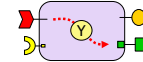
Lecture Outline

- Motivation for Middleware and Components
- Broad themes of Cadena
- A real-world test-bed from the avionics domain
- Main features of Cadena
 - component development
 - lightweight semantic annotations
 - intra-component dependencies
 - intra-component transition semantics
 - system assembly
- Analysis, automated design device, analysis driven configuration and customization of middleware and services
- Extending Bogor's modeling language to support Cadena designs
- Customizing Bogor's scheduling and state-space search modules to Cadena/BoldStroke designs

Leveraging Dependence Info

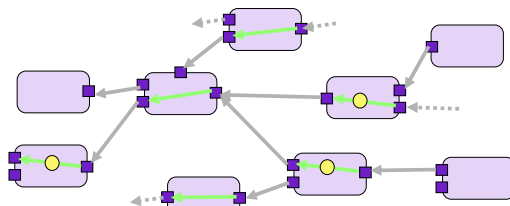
Dependence declarations are leveraged in a variety of ways...

state-based dependencies



- Form answers to visual queries about paths/dependences through configured systems
- Provides info to automatic/smart placement of pieces of KSU Event Communication Framework middleware service
- Basis of a number of forms of automated design advice (rate seeding, component distribution, etc.)

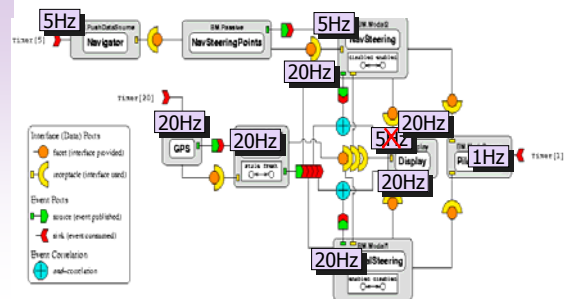
CCM Design Dependence Graphs



- From system configuration information
- From user-specified intra-component dependencies
- State predicates giving conditional dependencies

Aspect Synthesis

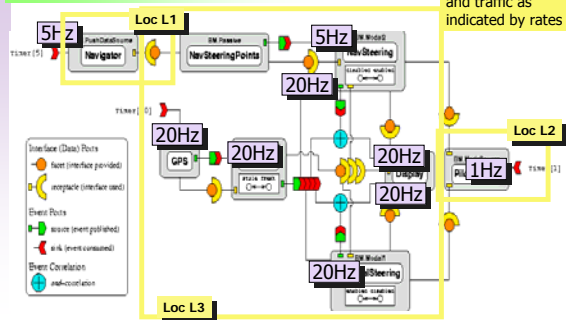
Dependency-driven rate assignment to event handlers



Aspect Synthesis

Synthesis of distribution information

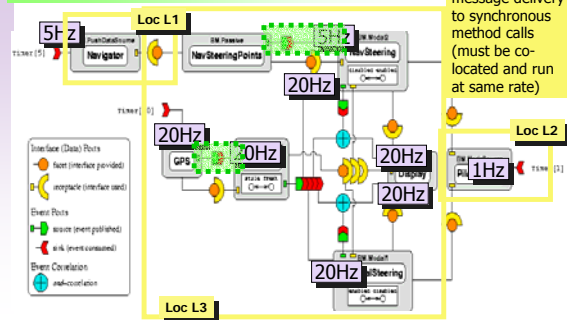
Look at coupling and traffic as indicated by rates



Aspect Synthesis

Automatic detection of optimization opportunities

Asynchronous message delivery to synchronous method calls (must be co-located and run at same rate)



Optimizing Event Communication

Common Situation

mode logic

If (mode = enabled), then fetch data, combine data and propagate.

If (mode = disabled), then ignore incoming events

group of sensors

Observe: if component is disabled, then event delivery of events from sensors is causing unnecessary overhead

...but all events flow through event channel, so...

Optimizing Event Communication

Move mode logic into event channel

group of sensors

mode logic

If (customer.mode = enabled), then forward event to consumer

If (customer.mode = disabled), then drop event

...generate customized event channels from high-level specifications

Event Channel

Configurable Product Line Profiles

Component type attributes
...master/proxy

Component instance attributes
...distribution location

Port attributes
...rate/priority

Connection attributes
...ERM
...event-communication layer

Profile-specific plug-ins

Boeinger JTRS Profile

Cadena profiles enable flexible definition of attributes for CCM model entities and APIs for plug-in tools to access and manipulate attribute values

CCM Development Support

Control:
API for plugging CCM frameworks into Cadena for modeling, design, and analysis.

Data:
Configuration data for D & C, Event Communication, etc.

CIAO (C++)

OpenCCM (Java)

XML Configuration

Zen RT ORB

CIAO Support

```

properties::connection {
// ===== OEP CCM =====
...
// ===== CIAO =====

optional("entry_point", STRING);
// Application-defined string that uniquely identifies the operation.
optional("worst_case_execution_time", INT);
// Execution times
optional("typical_execution_time", INT);
optional("cached_execution_time", INT);
// To account for server data caching.
optional("period", INT);
// For rate-base operations, this expresses the rate. 0 means "completely
// passive", i.e., this operation only executes when called.

```

Cadena profile

...this info can be entered/synthesized at the modeling level

CIAO Support

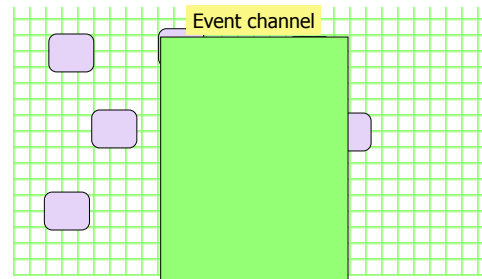
Auto-generate CIAO build files

CIDL Editor -- Skeleton generated automatically from CCM IDL

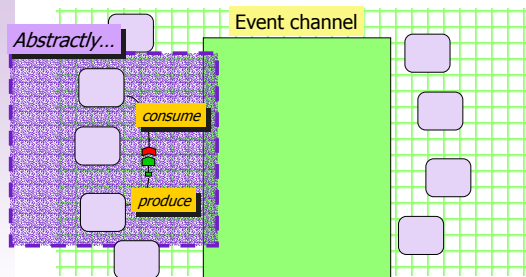
Lecture Outline

- Motivation for Middleware and Components
- Broad themes of Cadena
- A real-world test-bed from the avionics domain
- Main features of Cadena
 - component development
 - lightweight semantic annotations
 - intra-component dependences
 - intra-component transition semantics
 - system assembly
- Analysis, automated design device, analysis driven configuration and customization of middleware and services
- Extending Bogor's modeling language to support Cadena designs
- Customizing Bogor's scheduling and state-space search modules to Cadena/BoldStroke designs

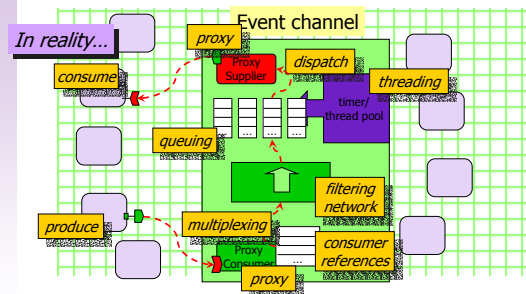
Boeing Threading Model



Boeing Threading Model

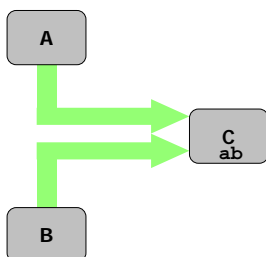


Boeing Threading Model



Challenges of Event Communication

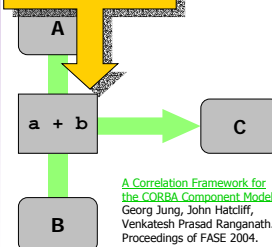
Consider the following situation:



- Component C is receiving from two components A and B
- A and B send at different rates
- C needs both inputs to become active

Challenges of Event Communication

If we add correlations to the infrastructure the following situation:



- We can:
- Reduce network traffic
 - Simplify computation inside the component
 - Clarify the design
 - Define the components in a more general way

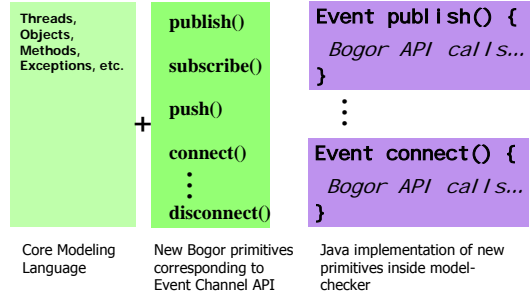
A Correlation Framework for the CORBA Component Model, Georg Jung, John Hatcliff, Venkatesh Prasad Ranganath. Proceedings of FASE 2004.

Middleware/Service Semantics

- Weak CCM and Event Services Specs (OMG)
 - Informal : English and examples
 - Intentionally under-specified to allow implementor freedom
- Looked at *implemented* semantics of existing ORBs and Event Services
 - ACE/TAO, FACET, OpenCCM, ...
- Developed a family of semantic models that captured their behavior
- Implemented these models as Bogor extensions
 - model modules are *reused* each time we reason about a system

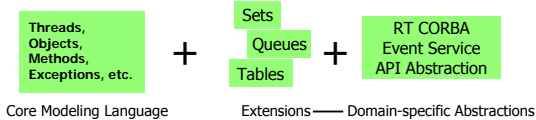
Domain-Specific Modeling

Bogor -- Extensible Modeling Language

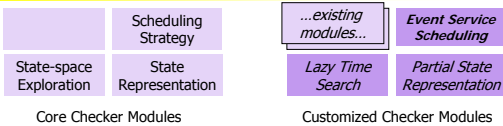


Bogor Customized To Cadena

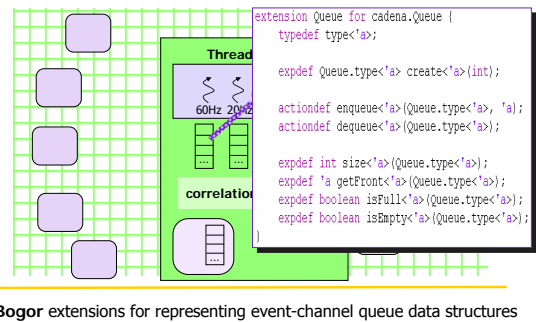
Bogor -- Extensible Modeling Language



Bogor -- Customizable Checking Engine Modules



Bogor Modeling Extensions



Bogor Modeling Extensions

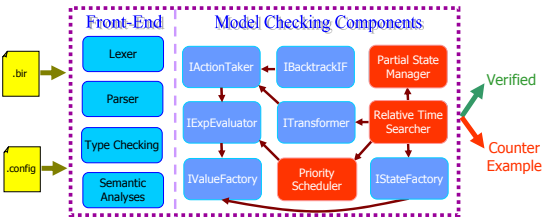


Lecture Outline

- Motivation for Middleware and Components
- Broad themes of Cadena
- A real-world test-bed from the avionics domain
- Main features of Cadena
 - component development
 - lightweight semantic annotations
 - intra-component dependences
 - intra-component transition semantics
 - system assembly
- Analysis, automated design device, analysis driven configuration and customization of middleware and services
- Extending Bogor's modeling language to support Cadena designs
- Customizing Bogor's scheduling and state-space search modules to Cadena/BoldStroke designs

Domain-Specific Algorithms

Bogor -- Customizable Checking Engine Modules



Bogor default modules are *unplugged* and *replaced* with state representation, scheduling and search strategies *customized to the Bold Stroke domain*

Assessments of Previous Work

Cadena	dSPIN (1CSE'02)	Bogor (FMCO'02)
Boeing ModalSP <ul style="list-style-type: none"> 3 rate groups 8 components 125 events per hp 	1.4 M states 58 sec 130 MB	9.1 K states 8.59 sec 1.61 MB
Boeing MediumSP <ul style="list-style-type: none"> 2 rate groups 50 components 820 events per hp 	X	740 K states 3 min 21.5 MB

- want to check larger model
 - does not seem to scale well regardless aggressive reductions

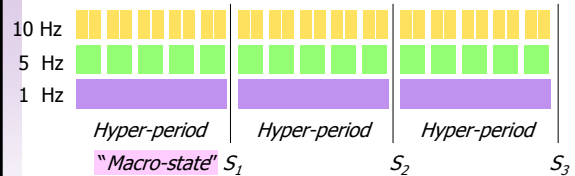
Key Observation

Leverage patterns of periodic computation

- use the structure of periodic systems to systematically drop states

Leveraging Periodic Structure

Periodic Tasks

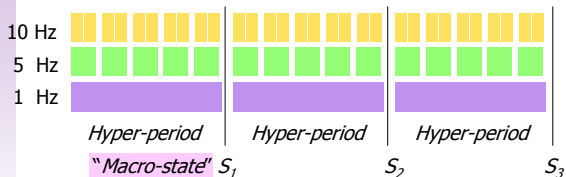


Basic Idea

- break the search into several regions
- divide the problem into smaller problems

Leveraging Periodic Structure

Periodic Tasks

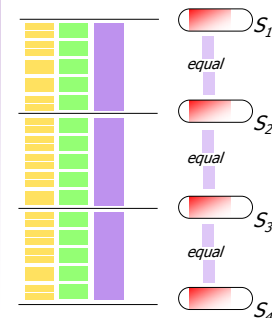


Macro-state Structure

- Same at each macro-state:
 - dispatch queues empty, threads idle, correlators are at initial state
- Different: component/system mode values are different

Quasi-Cyclic Structure

Trace Structure Macro-states



These successive macro-states maybe different (acyclic)...

...but a portion of each of the states is repeating...

...and so we say that the state-space is *quasi-cyclic*.

Quasi-Cyclic Structure

Trace Structure

Φ-states

$\Phi_{S_1}^{\text{conforming}}$
equal

$\Phi_{S_2}^{\text{conforming}}$
equal

$\Phi_{S_3}^{\text{conforming}}$
equal

$\Phi_{S_4}^{\text{conforming}}$

Generalizing

- Many applications with *control-loops* have this property
 - GUIs, web-servers,...
- Use a predicate Φ to characterize the repeating portion

Φ-Bounded Search

Trace Structure

Global State Store

Φ_0

Place initial Φ -state in global store, and begin state exploration.

Region State Store

Φ-Bounded Search

Trace Structure

Global State Store

Φ_0

Region State Store

Place states in region state store until Φ -state is encountered.

Φ-Bounded Search

Trace Structure

Global State Store

Φ_0, Φ_1

Place Φ -state into global store

Region State Store

Φ-Bounded Search

Trace Structure

Global State Store

Φ_0, Φ_1

Region State Store

Flush region state store

Φ-Bounded Search

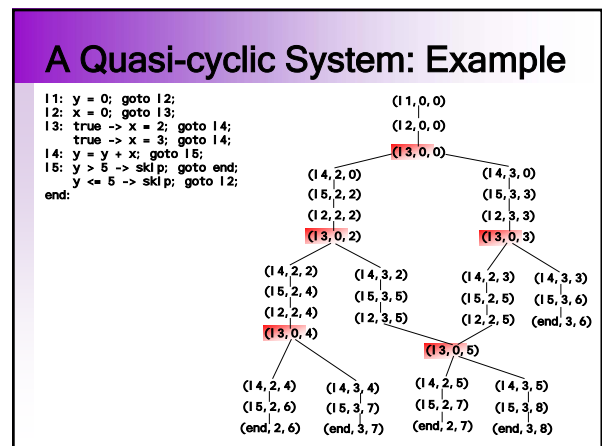
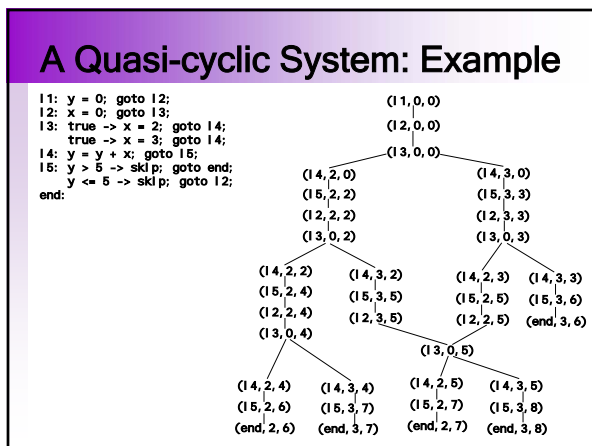
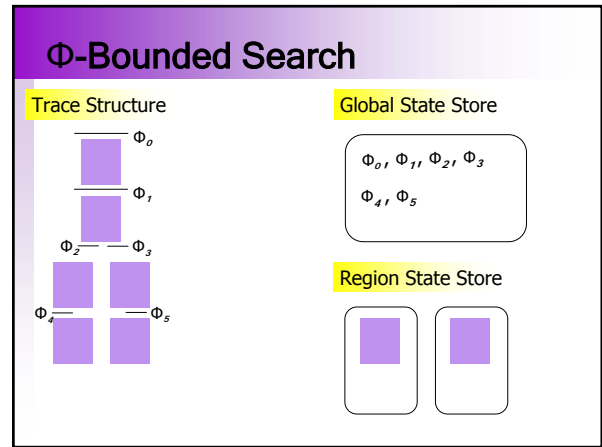
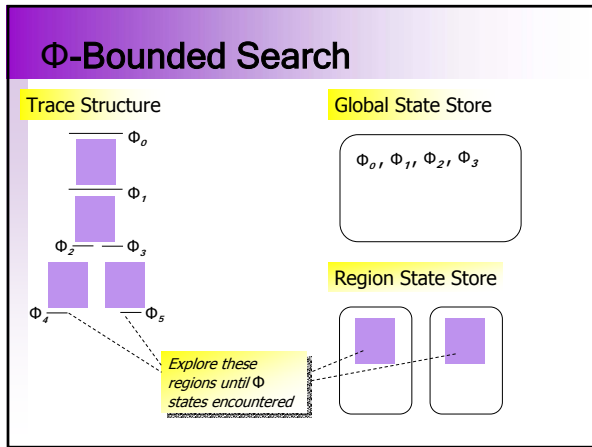
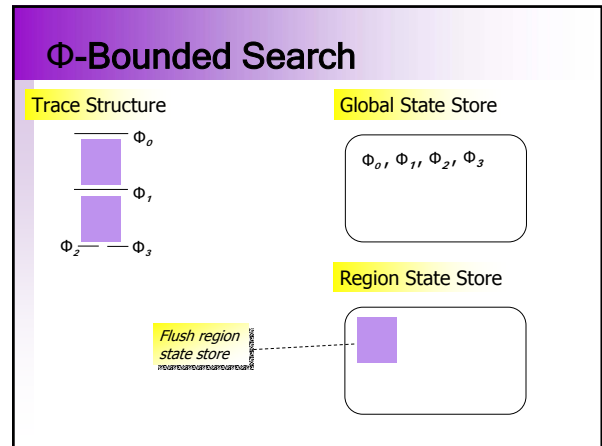
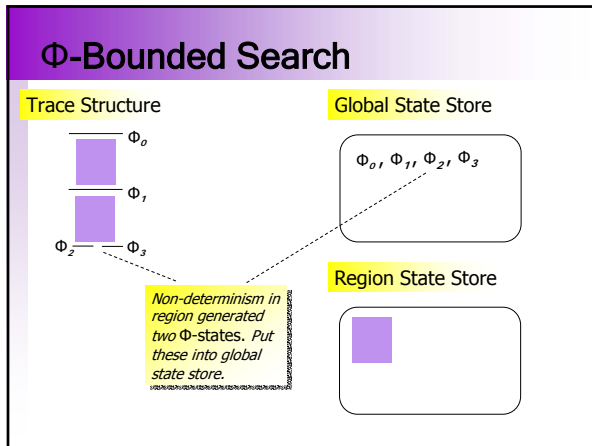
Trace Structure

Global State Store

Φ_0, Φ_1

Region State Store

Place states in region state store until Φ -state is encountered.



Quasi-cyclic Search: Example

(1, 0, 0)

$\Phi: pc = I3 \wedge x = 0$

Global States = {}

Queues = {}

```

11: y = 0; goto 12;
12: x = 0; goto 13;
13: true -> x = 2; goto 14;
    true -> x = 3; goto 14;
14: y = y + x; goto 15;
15: y > 5 -> skip; goto end;
    y <= 5 -> skip; goto 12;
end:
    
```

Quasi-cyclic Search: Example

(1, 0, 0)

(12, 0, 0)

(13, 0, 0)

$\Phi: pc = I3 \wedge x = 0$

Global States = {}

Queues = {0}

```

11: y = 0; goto 12;
12: x = 0; goto 13;
13: true -> x = 2; goto 14;
    true -> x = 3; goto 14;
14: y = y + x; goto 15;
15: y > 5 -> skip; goto end;
    y <= 5 -> skip; goto 12;
end:
    
```

Quasi-cyclic Search: Example

(1, 0, 0)

(12, 0, 0)

(13, 0, 0)

(14, 2, 0)

(15, 2, 2)

(12, 2, 2)

(13, 0, 2)

(14, 3, 0)

(15, 3, 3)

(12, 3, 3)

(13, 0, 3)

$\Phi: pc = I3 \wedge x = 0$

Global States = {0}

Queues = {2,3}

```

11: y = 0; goto 12;
12: x = 0; goto 13;
13: true -> x = 2; goto 14;
    true -> x = 3; goto 14;
14: y = y + x; goto 15;
15: y > 5 -> skip; goto end;
    y <= 5 -> skip; goto 12;
end:
    
```

Quasi-cyclic Search: Example

(1, 0, 0)

(12, 0, 0)

(13, 0, 0)

(14, 2, 0)

(15, 2, 2)

(12, 2, 2)

(13, 0, 2)

(14, 2, 2)

(15, 2, 4)

(12, 2, 4)

(13, 0, 4)

(14, 3, 0)

(15, 3, 3)

(12, 3, 3)

(13, 0, 3)

(14, 3, 2)

(15, 3, 5)

(12, 3, 5)

(13, 0, 5)

$\Phi: pc = I3 \wedge x = 0$

Global States = {0,2}

Queues = {3,4,5}

```

11: y = 0; goto 12;
12: x = 0; goto 13;
13: true -> x = 2; goto 14;
    true -> x = 3; goto 14;
14: y = y + x; goto 15;
15: y > 5 -> skip; goto end;
    y <= 5 -> skip; goto 12;
end:
    
```

Quasi-cyclic Search: Example

(1, 0, 0)

(12, 0, 0)

(13, 0, 0)

(14, 2, 0)

(15, 2, 2)

(12, 2, 2)

(13, 0, 2)

(14, 2, 2)

(15, 2, 4)

(12, 2, 4)

(13, 0, 4)

(14, 3, 0)

(15, 3, 3)

(12, 3, 3)

(13, 0, 3)

(14, 2, 3)

(15, 2, 5)

(12, 2, 5)

(13, 0, 5)

$\Phi: pc = I3 \wedge x = 0$

Global States = {0,2,3}

Queues = {4,5}

```

11: y = 0; goto 12;
12: x = 0; goto 13;
13: true -> x = 2; goto 14;
    true -> x = 3; goto 14;
14: y = y + x; goto 15;
15: y > 5 -> skip; goto end;
    y <= 5 -> skip; goto 12;
end:
    
```

Quasi-cyclic Search: Example

(1, 0, 0)

(12, 0, 0)

(13, 0, 0)

(14, 2, 0)

(15, 2, 2)

(12, 2, 2)

(13, 0, 2)

(14, 2, 2)

(15, 2, 4)

(12, 2, 4)

(13, 0, 4)

(14, 3, 0)

(15, 3, 3)

(12, 3, 3)

(13, 0, 3)

(14, 2, 3)

(15, 2, 5)

(12, 2, 5)

(13, 0, 5)

$\Phi: pc = I3 \wedge x = 0$

Global States = {0,2,3,4}

Queues = {5}

```

11: y = 0; goto 12;
12: x = 0; goto 13;
13: true -> x = 2; goto 14;
    true -> x = 3; goto 14;
14: y = y + x; goto 15;
15: y > 5 -> skip; goto end;
    y <= 5 -> skip; goto 12;
end:
    
```

Quasi-cyclic Search: Example

$\Phi: pc = I3 \wedge x = 0$

Global States = {0,2,3,4,5}

Queues = {}

```

11: y = 0; goto I2;
12: x = 0; goto I3;
13: true -> x = 2; goto I4;
    true -> x = 3; goto I4;
14: y = y + x; goto I5;
15: y > 5 -> skip; goto end;
    y <= 5 -> skip; goto I2;
end:
  
```

Quasi-cyclic Search: Example

$\Phi: pc = I3 \wedge x = 0$

Global States = {0,2,3,4,5}

Queues = {}

```

11: y = 0; goto I2;
12: x = 0; goto I3;
13: true -> x = 2; goto I4;
    true -> x = 3; goto I4;
14: y = y + x; goto I5;
15: y > 5 -> skip; goto end;
    y <= 5 -> skip; goto I2;
end:
  
```

Quasi-cyclic Search: Example

Quasi-cyclic Search: Example

- Search each region independently
 - max of 9 versus 37 states in classical DFS
 - note that the sum here is >37
 - same states may appear in multiple regions
- Regions can be searched in parallel
- Works well when
 - reasonable fraction of state variables are cyclic
 - low-degree of overlapping between regions

Domain-Specific Algorithms

Bogor -- Customizable Checking Engine Modules

Bogor default modules are *unplugged* and *replaced* with state representation, scheduling and search strategies *customized to the Bold Stroke domain*

Scaling Boeing ModalSP

- both searches have exponential time growth
 - quasi-cyclic search takes more time (overlapping regions)
- parallel quasi-cyclic takes 25% less time than classical DFS
 - actively pursuing distributed solution

Conclusions

- Model-driven component-based development provides a variety of benefits
- One goal of system architecture design is to lift as much aspect logic up to modeling level as possible
 - System integrator (who assembles components together to form a system) plays a crucial "programming role" by selecting/configuring attributes and services
- Bogor can be customized to check Cadena system designs
 - customized scheduling and search strategies
 - new BIR extensions model the APIs of component infrastructure and RT-CORBA event channel

For More Information...



SAnToS Laboratory,
Kansas State University
<http://www.cis.ksu.edu/santos>



Cadena Project
<http://cadena.projects.cis.ksu.edu>

...see us, for demo, examples, plug-in development, etc.