Real-Time Control of Large Structures

Ahmed Sameh, Christoph Hoffmann, Ananth Grama, Purdue University

Dan Sorensen, Thanos Antoulas
Rice University

Kyle Gallivan, Paul van Dooren
Florida State University, UCL-Belgium.

Supported by the National Science Foundation through the ITR/DDDAS program.
Project Goals

• Real-time control of structures through:
  – Sensing
  – Model Reduction and Control
  – Actuation
• High-Fidelity Modeling of Structures
• Comprehensive Validation
Model Reduction and Control

Bowen Model
n = 4950 DOF's
Model Reduction and Control
## Model Reduction and Control

<table>
<thead>
<tr>
<th>Index</th>
<th>$r = 62$</th>
<th>$r = 40$</th>
<th>$r = 20$</th>
<th>$r = 212$</th>
<th>$r = 80$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$J_1$</td>
<td>0.84169</td>
<td>0.83478</td>
<td>0.84207</td>
<td>0.81745</td>
<td>1.015</td>
</tr>
<tr>
<td>$J_2$</td>
<td>0.89064</td>
<td>0.91473</td>
<td>0.89061</td>
<td>1.0376</td>
<td>0.9938</td>
</tr>
<tr>
<td>$J_3$</td>
<td>0.90873</td>
<td>0.90204</td>
<td>0.90288</td>
<td>14.464</td>
<td>1.0693</td>
</tr>
<tr>
<td>$J_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$J_{12}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$J_{13}$</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>$J_{14}$</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>$J_{15}$</td>
<td>62</td>
<td>40</td>
<td>20</td>
<td>212</td>
<td>80</td>
</tr>
</tbody>
</table>
Model Reduction and Control

![Graph showing relative displacement of the roof over time with different reduced models and benchmark, using SVD based methods.](image)
Model Reduction and Control
Model Reduction and Control
High-Fidelity Modeling
Validation Testbed: Bowen Lab for Structural Engineering
Validation Testbed: Bowen Lab for Structural Engineering
Building a Robust Sensing Infrastructure
Demo

- COSMOS = mPL + mOS
  - Declarative *macroprogramming* of “aggregate network”
Demo

mPL program1

- timer(100) → accel
- accel → plotter

mPL program2

- timer(100) → accel
- accel → gate, max
- max → controller
- gate → plotter
- controller → gate
Talk Outline

• Macroprogramming heterogeneous networks using **COSMOS** (EuroSys’07)

• Building verifiable adaptive system components for sensornet applications using **Decal** (SOSP’07, submitted)
Wireless Sensor Networks

• How does one engineer robust and efficient sensing and control applications?
  – Resource constrained nodes
    • Network capacity
    • Memory
    • Energy
    • Processing capability
  – Dynamic conditions
    • Communication losses
    • Network self-organization
    • Sensor events
  – Performance and scalability requirements
  – Heterogeneous nodes
Programming Sensor Networks

- **Traditional approach:**
  - “Network enabled” programs on each node
  - Complex messaging encodes network behavior
  - No abstraction of low-level details
Programming Models

• Need high-level programming models that support
  • Flexibility and Expressiveness
    – Adding new processing operators
    – Routing – tree, neighborhood, geographical
    – Domain-specific optimizations (reliability, resilience, …)
    – …
  • Performance
    – Footprint of runtime support
    – High data rates
    – …
  • Robustness
The COSMOS Macroprogramming Environment

Program the network as a whole by specifying aggregate system behavior
Programming Model Overview

• Describe aggregate system behavior in terms of dataflow and data processing
• A program is an interaction assignment of functional components
• Low-overhead primitive dataflow semantics
  • Asynchronous
  • Best effort
Programming Model Overview

- Factor applications into reusable functional components
  - Data processing components
  - Transparent abstractions over low-level details
Programming Model Overview

- **Program correctness**
  - Assign types to inputs and outputs of FCs
    - Required interface and provided interface
  - Type checking

![Diagram]

- `accel`
- `timer`
- `max`
- `controller`
- `FFT`
- `display`
- `raw_t` to `max_t` to `fft_t`
Programming Model Overview

- **Heterogeneity**: Uniform programming model
- Capability-aware abstractions
- Aggregation primitives for nodes

@ sensor node:
- accel
- filter
- max

@ fast cpu:
- FFT

@ unique server:
- display
- controller
The COSMOS Suite

• Consists of
  – mPL programming language
    • Aggregate-system behavioral specification
  – mOS operating system
    • Low-footprint runtime for macroprograms
    • Dynamically instantiates applications
    • Node capability aware
  – Compilation infrastructure

• Current implementation supports Mica2 and POSIX platforms
Functional Components

- Elementary unit of execution
  - Safe concurrent execution
    - No side-effects from interrupts (on motes): No locks
    - Multi-threading on resource rich nodes (POSIX): Performance
- Static state memory
  - Prevents non-deterministic behavior due to malloc failures
  - Leads to a lean memory management system in the OS
- Dynamic dataflow handled by mOS
- Reusable components
Functional Components

- **Programmatically:**
  - Declaration in mPL language (pragma)
  - C code that implements the functionality
- For an application developer only the declaration is important
  - Assumes a repository of implementation
  - Binaries compiled for different platforms

```c
%fc_dplex: {
    fcid = FCID_DPELX,
    mcap = MCAP_SNODE,
    in [ raw_t, raw_t ],
    out [raw_t]
};
```
Functional Components – C code

```c
static fc_header_t fc_hdr DECLARE_HDR = {
    fcid: FCID_FFT,
    state_sz: sizeof(fft_state_t)
};

int cp_return_t
fft_fc(cp_param_t *param, bq_node_t *pbqn, num_conn_t ind)
{
    kiss_fftr_cfg pc; // using kiss fft implementation
    int nfft;
    uint8_t *d;
    nfft = GET_DATA_LEN_CRAW(pbqn);
    d = GET_DATA_POINTER_CRAW(pbqn);
    for (i=0; i<nfft; i++) {
        pfs->in[i] = d[i]; // copy uchar to in type.
    }
    pc = kiss_fftr_minit(nfft, 0, pfs->fft_cfg, CFG_SZ);
    if (pc != NULL) {
        kiss_fftr(pc, pfs->in, pfs->out_freq.data);
        pfs->out_freq.id = GET_ID_CRAW(pbqn);
        out_data(param, 0/*out index*/, &(pfs->out_freq),
            ID_SZ+sizeof(kiss_fft_cpx)*(nfft/2+1));
    }
    return FC_RETURN_OK;
}
```
Functional Components

• **Extensions: transparent abstractions**
  - Service FCs (SFCs)
  - Language FCs (LFCs)

• **SFCs**
  - SFCs implement system services (e.g., network service) and exist independent of applications.
  - May be linked into the dataflow at instantiation

• **LFCs**
  - Used to implement high-level abstractions that are spliced into the IA transparently during compile time
mPL: Interaction Assignment

timer(100) → accel
accel → filter[0], smax[0]
smax[0] → controller[0] | smax[1]
filter[0] → fft[0]
fft[0] → display
controller[0] → filter[1]
Application Instantiation

- Compiler generates an annotated IA
- Each node receives a subgraph of the IA (based on capability)
  - Over the air (re)programming
- Local dataflow established by mOS
- Non-local FC communication bridged through the network SFC
Programming Model Overview

- Routing is simply a transparent SFC
  - Instantiates distributed dataflow
  - For example, hierarchical tree routing
High-Level Abstractions

• **Appear as contracts on dataflow paths**
  • Enables semantics of *dataflow bus*
• **No modifications to runtime or compiler**
• **Syntax:**
  • Name property
    =`const`, `[in:stream, ...] [exception:stream, ..] [out:stream, ...]`
• **Implemented in LFCs identified by Name-property pair**
High-Level Abstractions

• Example: Type translation adapter:
  • buffering to reassemble packets based on time-window: buffer LFC

• \texttt{filter[0] \rightarrow (buffer \ line=FFT\_LEN \ out:fft[0])}
High-Level Abstractions

- Other Examples: Dataflow path QoS.
  - Reliability of packets
  - Priority of data
  - Forward error correction, channel coding
  - Congestion avoidance/mitigation
Low-footprint Dataflow Primitive

• Data driven model

• Runtime Constructs:
  – Data channels implemented as output queues:
    • No single message queue bottleneck → concurrency
    • No runtime lookups and associated failure modes
    • Minimizes memory required by queue data

• Constructs for synchronization of inputs
Efficient Concurrency and Locking

- **mOS** supports both multi-threaded and non-preemptive environments
- **Motes** have non-preemptive scheduling
  - Locking through disabling interrupts
- **On resource rich nodes**
  - Multi-threading: concurrent FC execution
  - **Scope of locks**: interacting FCs
Experimental Evaluation

- Micro evaluation for Mica2
- CPU overhead

![Graph showing CPU utilization vs. sampling rate](image)

- Higher CPU utilization than TinyOS
- Slightly better than SOS
- COSMOS provides macroprogramming with a low footprint
- Plus other significant optimizations
Experimental Evaluation: Heterogeneous Scaling

• Experimental Setup:
  – 1 to 10 Mica2 motes
    • (7.3MHz, 4KB RAM, 128KB P-ROM, CC1K radio)
  – 0 to 2 Intel ARM Stargates
    • (433MHz SARM, 64MB SDRAM, Linux 2.4.19, 802.11b + Mica2 interface)
  – 1 workstation
    • (Pentium 4, Linux 2.6.17, 802.11a/b/g + Mica2 interface)
Experimental Evaluation

• Heterogeneous networks yield higher performance

- Decreased latencies
- Less congestion
- Fewer losses (shorter distances)
- COSMOS enables performance by vertical integration in heterogeneous networks
- ... And ease of programming
Experimental Evaluation

- Reprogramming:
- IA description sent as messages
  - Describing each component and its connections

<table>
<thead>
<tr>
<th>Component name</th>
<th>Machine capability</th>
<th>Size (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>accel</td>
<td>MCAP_ACCEL_SENSOR</td>
<td>12</td>
</tr>
<tr>
<td>cress</td>
<td></td>
<td></td>
</tr>
<tr>
<td>threshold</td>
<td></td>
<td></td>
</tr>
<tr>
<td>max</td>
<td></td>
<td></td>
</tr>
<tr>
<td>max</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fft</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ctrl</td>
<td>NOT(MCAP_ACCEL_SENSOR) &amp; NOT(MCAP_UNIQUE_SERVER)</td>
<td>16</td>
</tr>
<tr>
<td>disp1</td>
<td>MCAP_FAST_CPU</td>
<td>11</td>
</tr>
<tr>
<td>disp2</td>
<td>MCAP_ANY</td>
<td>11</td>
</tr>
<tr>
<td>net(SFC)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>buff-len(LFC)-fft</td>
<td></td>
<td></td>
</tr>
<tr>
<td>buff-time(LFC)-max</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Size of messages is small ~ 11 to 13 Byte
→ Number of inputs and outputs

Only 10s of messages per application

COSMOS, by design, allows low-overhead reprogramming
Experimental Evaluation

- Effect of adding FCs

Adding FCs on the dataflow path has low-overhead: Supports highly modular applications through low-footprint management of FCs.
Verifiable Sensing Applications

- Sensor networks require long-term autonomic execution
- Network dynamics require adaptability
- Management of conflicting trade-offs
- Dependencies between components
Verifiable Applications Using DECAL

- Declarative concurrent programming model based on logic of actions
- Control over resource utilization through state abstractions
- Simplified programming of distributed protocols
- Temporal logic invariants to enable verification
- Verification of component interactions
Programming Model Overview

Application

uses

Hi-res-accel: stream component
Max-accel: stream component
Tree-route: service component
Programming Model Overview

- Applications can read and affect system state via a set of abstraction variables
- Each stream has an isolated view of system:
  - Allows independent development of streams
  - Enforce arbitration on the use of physical resources
- Actions cause state transitions.
  - They specify local node’s sensing, data processing, data forwarding behavior
- Guards trigger the action if a pre-condition is met by the state
Programming Model

- **Concurrent programming abstraction**
  - Actions are triggered *whenever* the pre-conditions are met
  - Logic of actions (Lamport)
  - We have developed protocols (routing, collaborative rate control) that span 100s of line of C code, in only 15-30 lines

- **System state abstractions enable development of adaptive applications**
  - Actions are enabled based on the live state of the system

- **High-level declarative programming**
  - Pre-conditions are logic expressions
  - Actions are *specified* in terms of state transitions: $S \rightarrow S'$
Programming Model

- **State abstraction selection**
  - Comprehensive list of information and control variables relevant to sensor network development

- **Key abstractions**
  - **Ingress and egress buffer management**
    - Observe load and congestion conditions
    - Observe dynamic routing organization, packet losses
    - Enables radio power management
    - Enables load conditioning, windowed buffering and aggregation, data prioritization
    - Ease of programming using whole buffer operations (there exists, for all) and logic expressions

- **Power management**
- **Packet transport (e.g., acknowledgements, attributes)**
Distributed Coordination Primitives

• Variable type : feedback variable
  • Writing to a variable of this type results in a local-wireless broadcast of the value
    - Var : FbVar(int a, int b) // declaration
    - If due to an action: Var != Var’ then broadcast value of Var’

• Obviates control messaging
  • Ported routing protocol from ~400 lines of C code to 20 lines

• Safety in implementing control protocols
• Low-overhead implementation
Declaring Actions

• **Generic guard** expressions

  \[
  \text{Guard( Tag, pre-conditions... , [filters,] post-conditions... ) : label}
  \]

• **Tag determines default pre and post conditions**

  • Common operations are abstracted
  • Example, processing (and removal) of data from ingress buffer, sorting buffers, handling feedback variables
  • Default post-conditions enforce correct usage and semantics
Declaring Actions

• Example Load-shedding:

Guard(ProcEB, \{photo\}, Len(EB) > 10, Remove(_) , UpdAvg(_, true);
Remove(a) \triangleq \exists b \in EB : a.tseq = b.tseq;
Guard(SortEB, \{photo\}, true, TimeOrder(_, _), true);
TimeOrder(a, b) \triangleq \text{SmallerSeq}(a.tseq, b.tseq);

• Example tree-routing:

parent : FbVar(int addr, int hops);
Guard(FbVar, parent, true, GotBeacon(remote.val));
GotBeacon(n) \triangleq \text{parent'} = (n.addr \neq \text{LocalId} \land n.hops < parent.hops) ?
[addr \mapsto \text{SrcNodeId}(n), \text{hops} \mapsto n.hops + 1];
Declaring Invariants

- Users can provide invariants using temporal logic expressions

\[
\text{invariant IBLenInv} \triangleq (\square (\text{Len}(\text{IB}) < 20)) \land (\Diamond (\text{Len}(\text{IB}) < 5));
\]

- Users can provide assertions on values

- Verification proves that a program meets invariants
Reusable Components

- Components implement policies and protocols
  - Exports invocable interface
  - Reads stream’s state variables
    » Enables concurrent execution
  - Exports local variables
  - Can implement control protocols using private local variables

- Example: congestion control component
  - Reads length of ingress and egress buffers
  - Exports the rate at which the local node should send data (rate control policy)
  - Uses private feedback variables to announce congestion to neighboring nodes (interference aware back pressure protocol)
  - If successors are congested decreases local rate (fair rate control protocol)
Composition Verification

- Components verify that its invocation behaves correctly
  - *Monitor* state variables
    - Pre-conditions to match violation and trigger corrective action or raise assertion failure
  - *Monitor* user actions using guard declaration tags
    - e.g., congestion control component: `monitor Guard(InsertEB, …)`
    - Monitor data insertions into the egress buffer and check that they respect the rate decided by fair rate control protocol

- Component users verify that the values exported by components meet application objective
  - e.g., congestion control user: `invariant □ (rate > min_rate)`
  - If minimum rate is required → use components that guarantee minimum rate
  - Correct component implementation: exponential back off + scheduling on-off periods of neighboring nodes

- Most “*monitor*” expressions required only during verification process ➔ Efficient runtime code
Program Synthesis

- User program
- Components
- TLA+ Spec
- TLC Model checker
- Synthesis Engine
- C code
- Runtime Library
- gcc
- Binary
Program Synthesis

• Verification by converting code to TLA+ and model-checking using TLC
  — Invariants specified as temporal logic expressions
  — Model checking checks by state space exploration
    • Liveness properties (e.g., pre-conditions are eventually true)
    • Safety properties (e.g., type checking, buffer overflows)
    • Decal language semantics (e.g., action post-conditions conformity)
    • User provided invariants, deadlock detection
  — Model checking is tractable
    • Domain-knowledge built in the DECAL language
    • Additional model reduction techniques
Code Generation and Runtime

- **Declarative programming can generate efficient platform optimized code**
  - Scheduling of enabled actions
    » e.g., prioritize data forwarding, load shedding
  - Inlining actions
    » Each action is a scheduled function
    » Minimize scheduling overheads by inlining actions

- **Key runtime isolation features**
  - **Network isolation**
    - Prevent high rate data applications from swamping low rate (yet critical messaging)
      » Each stream has independent ingress and egress buffer
      » Network stack provides round robin service
  - **Radio power management**
    - Radio is the key power consumer
    - Runtime provides isolation by prioritizing QoS of demanding flows even at higher energy cost
      » e.g., Low rate / low power during periods of uninteresting sensor data
      » High rate on data from high priority stream
Concluding Remarks

• This DDDAS Project has resulted in novel systems infrastructure critical to engineering complex sensing applications in diverse environments, while supporting programmability, scaling, performance, and verifiability.

• Forthcoming results: real-time support, robustness.
Related Work

- **TinyOS**
  - Low footprint: applications and OS are tightly coupled
    - Nice to program an OS, difficult to program applications
  - Scripting languages TinyScript*, Mottle*, SNACK
  - Maté – application specific virtual machine
    - Event driven bytecode modules run over an interpreter
    - Domain specific interpreter
    - Very low cost updates of modules
    - Easy to program using simple scripting languages*
    - Flexible? Expressive?
Related Work

- **TinyDB**
  - Specify system behavior using SQL queries
  - *It is a macroprogramming application*
  - Lack of flexibility
    - Adding delivery reliability?
    - Adding congestion control?
    - How to deal with perturbations?
    - Difficult to add new operators
  - Heavy footprint
    - Restricts scope to low-data rate apps
  - Highly coupled with routing
    - Resilience issues
Related Work

• High level macroprogramming languages
  – Functional and intermediate programming languages
    • Region stream, abstract regions, HOOD, TML (DTM)
  – Programming interface is restrictive and system mechanisms can not be tuned
  – No mature implementations exist, no performance evaluation is available
  – Compile down to native OS: can compile down to COSMOS applications
Related Work

• **SOS**
  – Interacting modules compose an application
  – OS and modules are loosely coupled
  – Modules can be individually updated: low cost
  – Larger number of runtime failure modes

• TinyOS and SOS are both node operating systems
WSN @ BOWEN

Pilot deployment at BOWEN labs

MICA2 motes with ADXL 202

Laser attached via serial port to Stargate computers

FM 433MHz

802.11b Peer-to-Peer

Currently laser readings can be viewed from anywhere over the Internet (conditioned on firewall settings)
Related Work

• **Dynamic resource allocation**
  - Impala
    • Rich routing protocols
    • Rich software adaptation subsystem
    • Aimed at resource rich nodes
  - **SORA**
    • Self-organized resource allocation architecture

  – Complimentary to our work
Programming Model Overview

• Over-the-air programming
  – Send components (borrowed from SOS)
  – Send IA subgraphs (low-overhead)
  – …and reprogramming
OS Design

- Each node has a static OS kernel
  - Consists of platform dependent and platform independent layers
- Each node runs service modules
- Each node runs a subset of the components that compose a macro-application
Implementation

- Implementations for Mica2 and POSIX (on Linux)

- **Mica2:**
  - Non-preemptive function pointer scheduler
  - Dynamic memory management

- **POSIX:**
  - Multi-threading using POSIX threads and underlying scheduler
  - The OS exists as library calls and a single management thread
FC-mOS Interaction

• FC parameters: cp_param_t, bq_node, index
  • Runtime parameters
  • Status of input queue

• Non-blocking system calls e.g., out_data()

• Return values
  – “commands” to manage the input queues
    • Commit
    • Transfer to output
    • Wait for more data (on same or different queue)
Base mPL LANGUAGE

- **Sections:**
  - Enumerations
  - Declarations
  - Instances
  - Mapping constraints
  - IA Description
  - Contracts
- **Implemented using Lex & Yacc**
Declarations

// declarations (auto import)
%accel_x : mcap = MCAP_ACCELSENSOR,
          device = ACCEL_X_SENSOR, out[raw_t];
%cpress_fc : { mcap = MCAP_ANY, fcid = FCID_CPRESS,
              in[raw_t], out[craw_t] };
%thresh_fc : { mcap = MCAP_ANY, fcid = FCID_THRESH,
              in[craw_t, ctrl_t], out[craw_t] };
%ctrl_fc : { mcap = MCAP_ANY, fcid = FCID_CTRL,
              in[max_t], out[ctrl_t] };
%max_fc : { mcap = MCAP_ANY, fcid = FCID_MAX,
              in[craw_t, max_t], out[max_t] };
%disp : mcap = MCAP_UNIQUE_SERVER,
        device = DISPLAY, in[*];
%fft_fc : { mcap = MCAP_FAST_CPU, fcid = FCID_FFT,
            in[craw_t], out[freq_t] };}
Instances & MCAP refinements

// logical instances
accel_x : accel(12);
disp : disp1, disp2;
cpress_fc : cpress;
thresh_fc : thresh(250);
max_fc : max;
fft_fc : fft;
ctrl_fc : ctrl;

// refining capability constraints
@ on_mote = MCAP_ACCEL_SENSOR : thresh, cpress;
@ on_srv = MCAP_UNIQUE_SERVER : ctrl;
WSN @ BOWEN

Pilot deployment at BOWEN labs

MICA2 motes with ADXL 202

Laser attached via serial port to Stargate computers

FM 433MHz

802.11b Peer-to-Peer

Currently laser readings can be viewed from anywhere over the Internet (conditioned on firewall settings)
Evaluation

• Remote sensing application
Extensibility

• Core of the OS is platform independent
• New devices can be easily added by implementing simple device port interface functions
• Communication with external applications by writing a virtual device driver
• Complex devices can use an additional service to perform low-level interactions
Evaluation

- Load conditioning
  - A 3pps FC on a processing node
  - 1pps per node received at processing node

Load conditioning enables efficient adaptation and self-organization in dynamic environments.
Example