Middleware Design in Networked Embedded Systems

Miklos Maroti and Akos Ledeczi

Institute for Software Integrated System
Vanderbilt University

Chess Review
May 8, 2003
Berkeley, CA

Overview

- Sample application: shooter localization
- Required middleware services
- Problems encountered
- Model-integrated computing for I/O Automata and TinyOS
- Formal verification
- Lessons learned

Application: Shooter localization

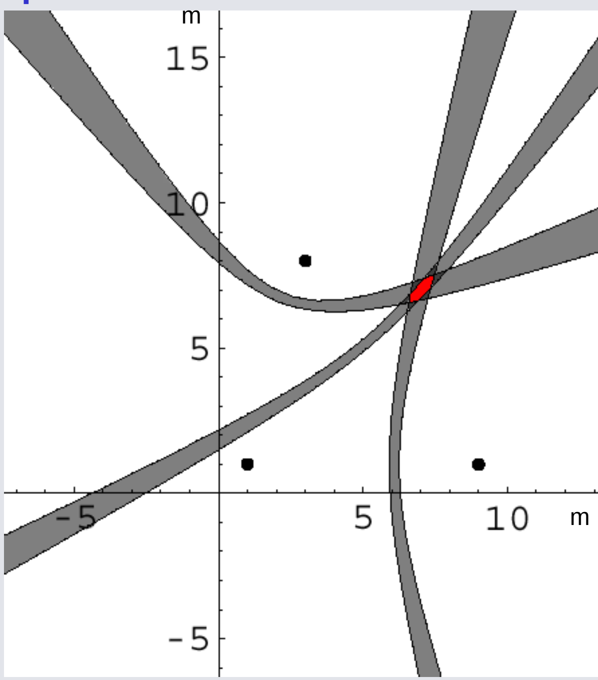
- Locate shooters in a large area using
  - a few hundred motes (Berkeley MICA motes)
    - 4 MHz microcontroller
    - 4 KB of RAM, 128 KB or ROM
    - Wireless communication, 4 KB/sec
    - Densely deployed at unknown locations
    - (cheap) microphone and buzzer
    - Runs on two AA batteries
  - and a base station (laptop or iPAQ)
- Application challenges:
  - Multiple shots, very noisy environment
  - Echoes from buildings, no line of sight
  - Supersonic weapons: shockwave and muzzle blast
  - Unknown bullet speeds
  - Isolation and grouping of “interesting” events

Algorithm: Detection

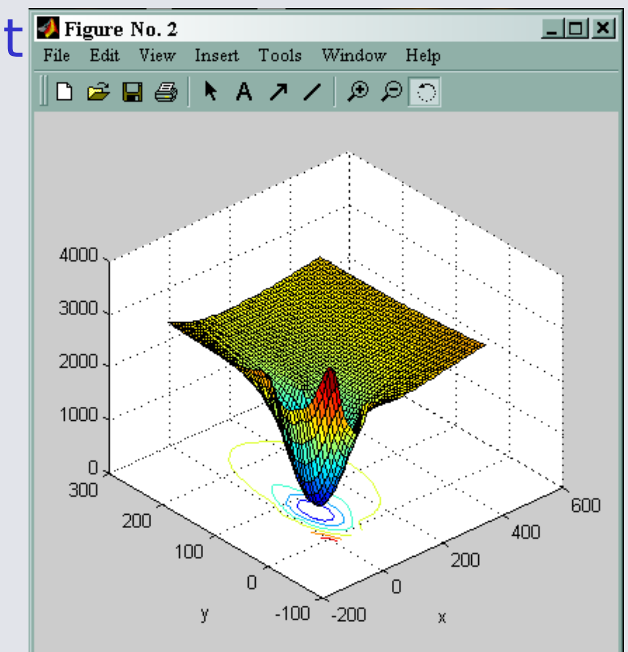
- 1-mic sensor board:
  - Measure time of arrival
  - Sample the microphone around 20 KHz and record data
  - If something loud happens then analyze data
  - Distinguish the shockwave and muzzle blast
  - Identify the weapon type/caliber using zero crossings
  - Determine the “leading” edge of the signal
  - Report the event to the base station
- 3-mic sensor board:
  - Measure direction of arrival
  - 3 microphone array (5 cm apart)
  - 1 MHz sampling
  - Processing by on board FPGA
  - Shockwave timestamp and pair wise TDOA

Algorithm: Localization

- The base station collects and analyzes events using
  - Time difference of arrival (TDOA)
  - Error surface (deviation of shot times from a given point)
  - ecting t



Triangulation error caused by 1 ms time synch error

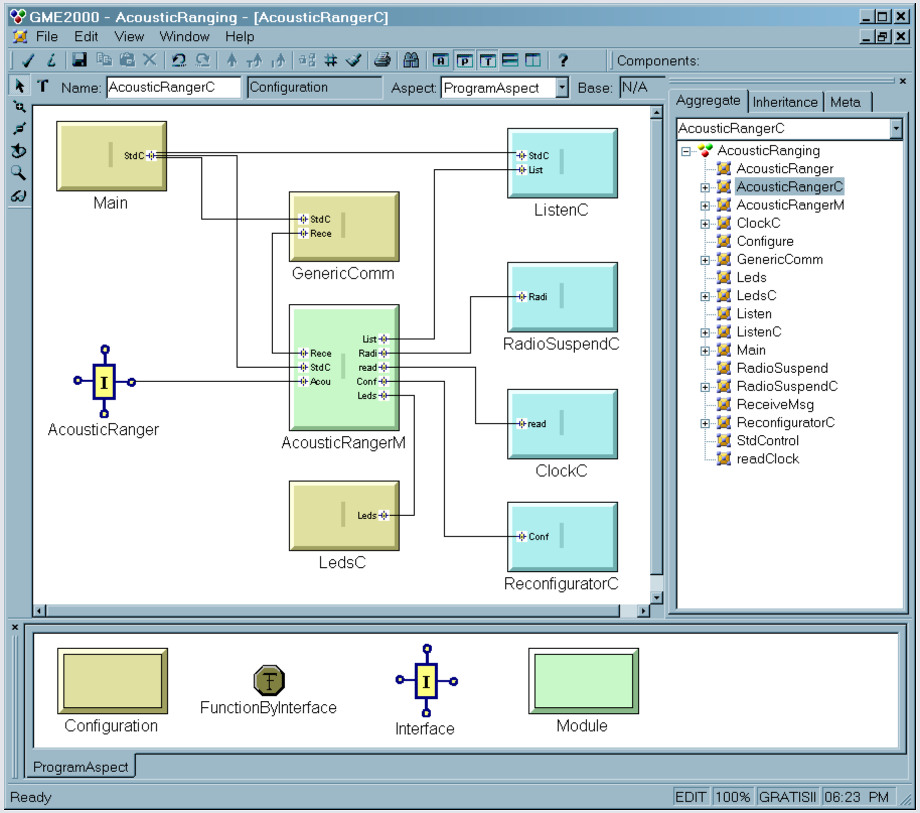


Necessary Middleware Services

- Application modeling language: Gratis II
  - graphical development environment for TinyOS
- Middleware services:
  - Multi-hop time synchronization with 1 ms of accuracy
  - Mote localization with 1 m of accuracy
  - Reliable message routing to the base station
- Local services:
  - Mote orientation using magnetometer
  - Microphone sampling at 20 KHz

TinyOS and Gratis II

- Everything is a component
- Components has provided and used interfaces, fixed memory frame
- Interfaces are bidirectional (commands and events)
- Application is a collection of statically wired interacting components
- Two level scheduling (non-preemptive): events and tasks
- Gratis II: automatic configuration generation



- TinyOS: Huge library of existing components and applications

Problems encountered

- Time synchronization:
  - We used the time of sendDone and receive events to establish synchronization points
    - Works fine in simple test application
    - Breaks when other interactive components are present (arbitrary delay)
    - Require OS support: timestamp in radio stack
  - When calculating linear regression we run into representation problems (float is not enough)
    - Knowing the possible range of values and the expected clock skew we can design better algorithm
  - How to implement and verify robustness
  - How to formally verify the accuracy

Problems encountered (2)

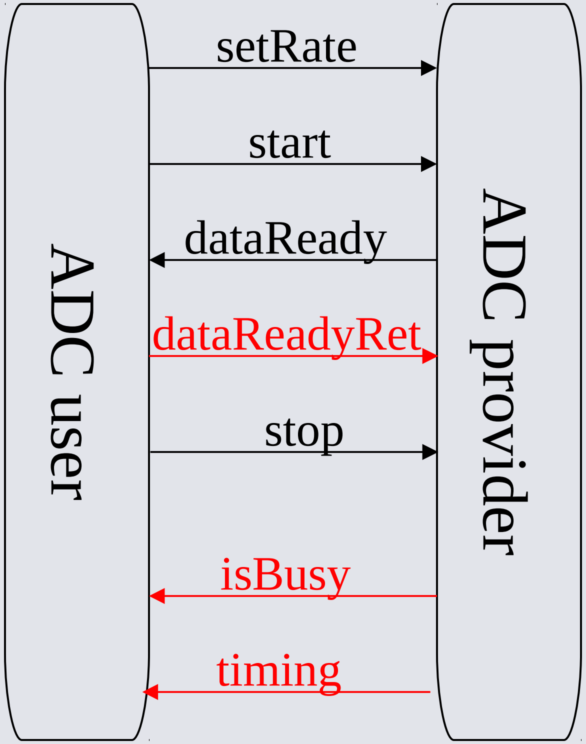
- Localization with acoustic ranging, idea:
  - "beacon" sends out a radio signal followed by an audio signal
  - "ranger" measures the time difference of arrival, calculate distance
- Modification needed because of noise:
  - "beacon" emits several audio signals
  - "ranger" records them and take the average
- Requirements:
  - Time synchronization (a single radio message can be used)
  - The "beacon" must buzz and the "ranger" must start recording at the same physical time
  - Fixed physical sampling rate for proper alignment
  - We must know the sampling frequency for digital filtering
- Testing:
  - Worked fine in simple test application
  - Broke when other components were present: samples sometimes arrived in the wrong order, and other timing problems

Problems encountered (3)

- Different platforms: MICA and MICA2
  - Have different core frequencies
    - different microphone sampling frequency
    - different buffer sizes and other constants
    - filters need to be redesigned, etc.
  - MICA2 has slightly better ADC
  - Different radio stack: the MICA2 uses the ADC to measure RSSI for collision avoidance.
    - We cannot sample the microphone and use the wireless communication at the same time
- Solution: turn alternatively on/off the radio stack and the microphone
  - Need to modify and verify the existing middleware services
  - New middleware service to coordinate the on/off phases on different motes

The ADC driver interface

- The TinyOS interface is functional
- Need behavioral interface, especially dataReadyRet
  - samples arrived in wrong order
  - buffer samples at ADC provider
- Need fixed sampling rate
  - time difference between two consecutive samples is in [r-ε,r+ε]
  - time difference between any two samples is in [nr-ε,nr+ε]
- When does the sampling start in real time?
  - Fixed delay from start
  - Delay is between bounds
- What is the timestamp of the sample
- Who else is using the ADC

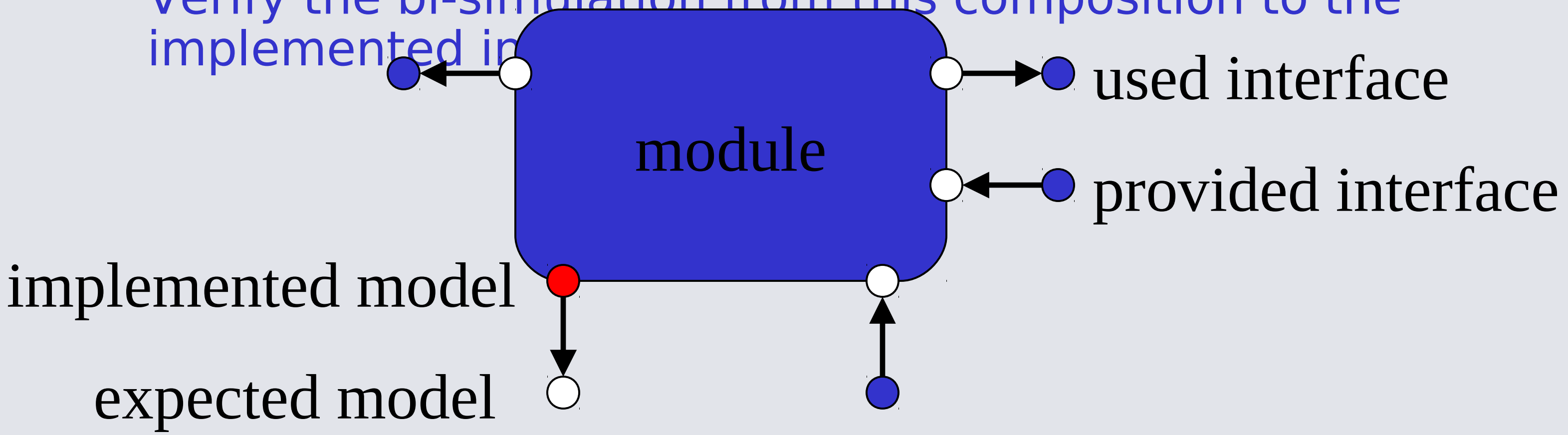


Component and composition verification

- Use I/O Automata
  - Existing tool chain developed at MIT (Nancy Lynch)
    - IOA language: nondeterministic, declarative
    - Safety and liveness properties
    - Forwards and backwards simulations
    - Invariants, and other assertions (first order language)
    - Composer, theorem prover (Larch), and simulator
    - Network size and topology is not limited
  - Exploit existing and verified distributed algorithms
- Model each interface using IOA
  - Two models: user and provider
  - Compose these two models
  - Specify and verify assertions

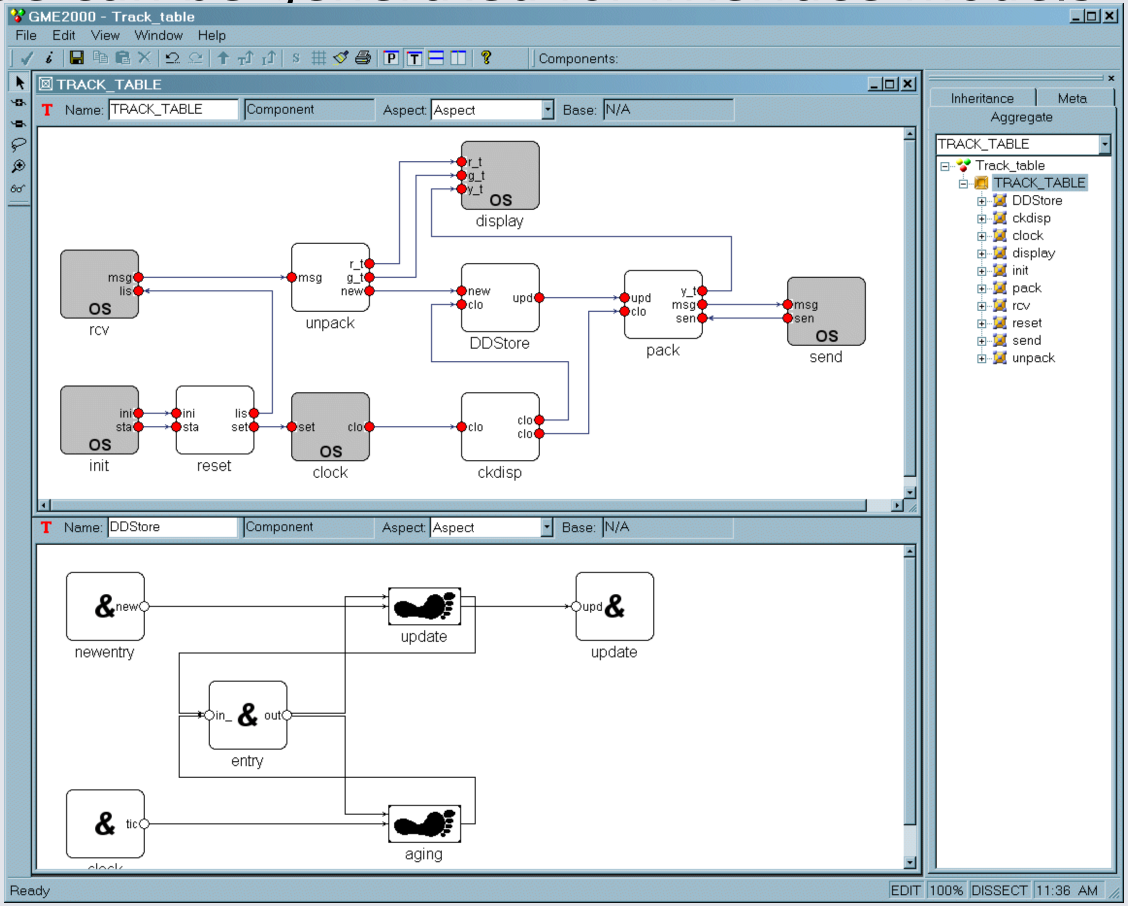
Module verification

- Model each module using IOA
  - Has used and provided interfaces
  - The interfaces have implemented and expected interface model
  - State properties that this implementation relies on
- Verify that the module implements its used and provided interface models using bi-simulation
  - For each implemented interface model (red dot), compose the module with all other expected interface models (blue dot).
  - Verify the bi-simulation from this composition to the implemented i



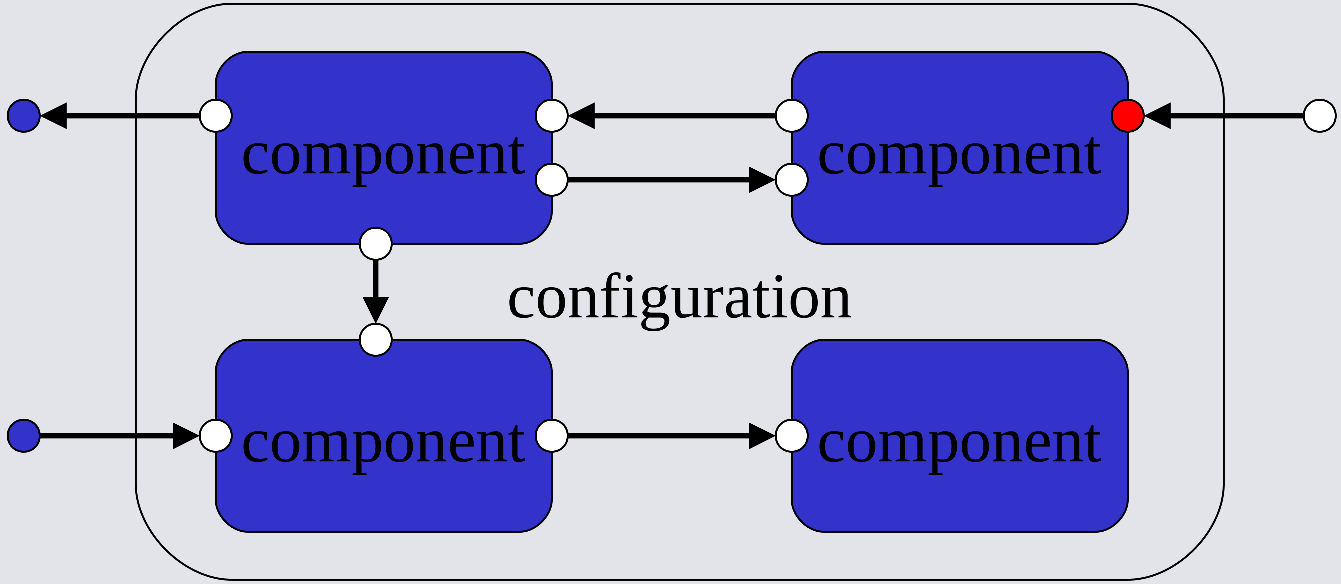
Code generation

- Executable code for simple modules can be generated from models
- Test code can be generated for interface models



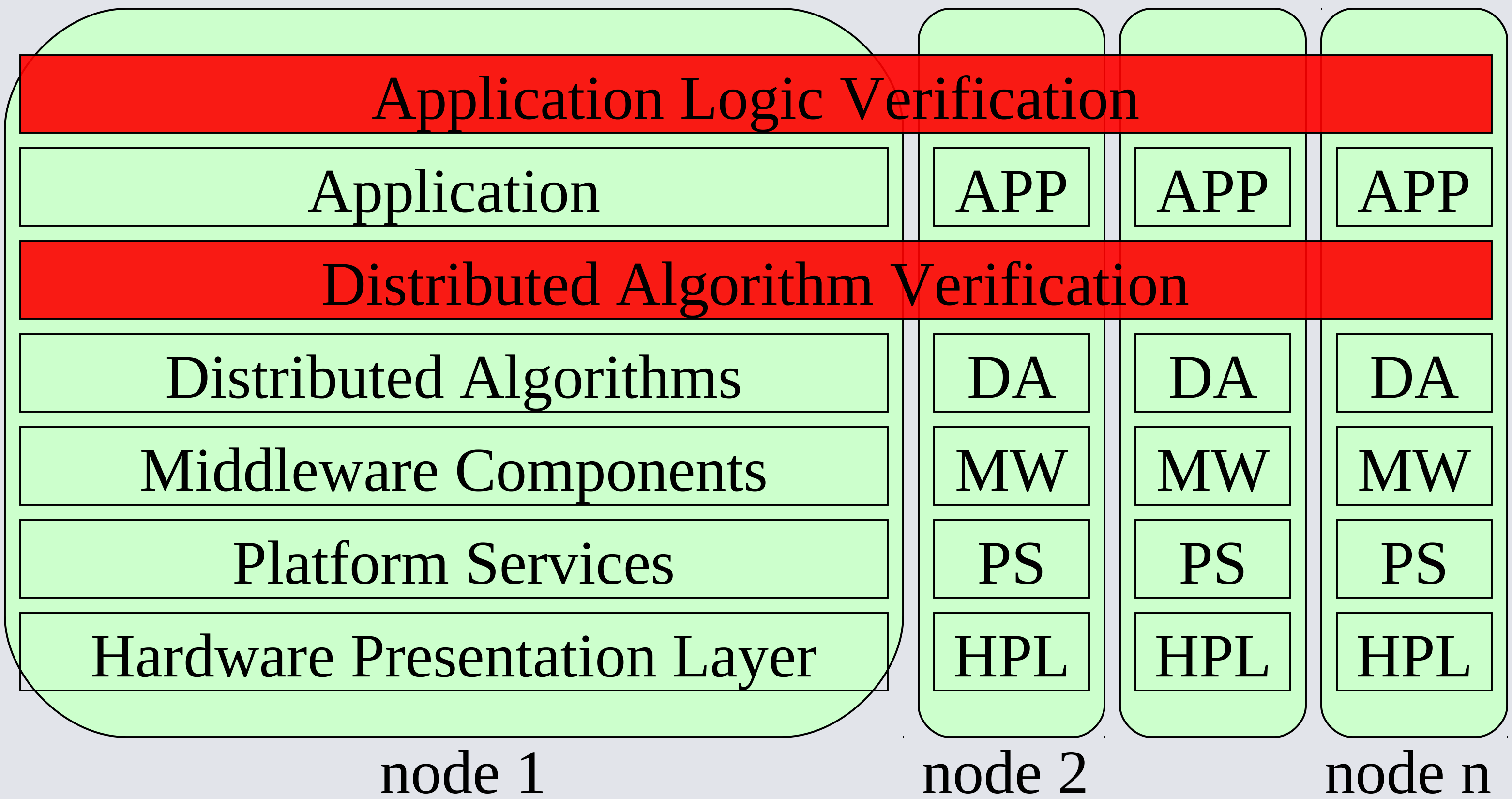
Configuration verification

- No new IOA models, just composition
- State and verify new invariants and properties
- Verify interface implementation



Different levels of abstraction

- Each horizontal box is full hierarchy of interfaces, modules and configurations.
- Not every level has executable code



Lessons Learned

- Modeling abstractions: resource constraints couple everything, deep modeling is needed
- Platform characteristics impact abstractions for application modeling
- Supporting more than one platform will help in getting the right abstractions
- Parameters and their interdependencies must become first class objects in models
- We are far from auto generating of code
- Experiment, experiment, experiment