Distributed Middleware Services Composition and Synthesis Technology

Miklós Maróti, Péter Völgyesi, Gyula Simon, Gábor Karsai and Ákos Lédeczi

Institute for Software Integrated Systems, Vanderbilt University

Outline

- Introduction
- Asynchronous I/O automata
- Structured sets and data ports
- Structural model of I/O automata
- Case study
  - TinyOS: cooperative acoustic tracking
  - Siesta: routing in vibration control simulation
- Conclusion

Networked Embedded Systems Technology (NEST)

- Large volume fabrication of compact autonomous nodes
- Have limited resources and communication capabilities
- Large number of densely deployed processing nodes
- Tightly coupled to physical processes

NEST applications

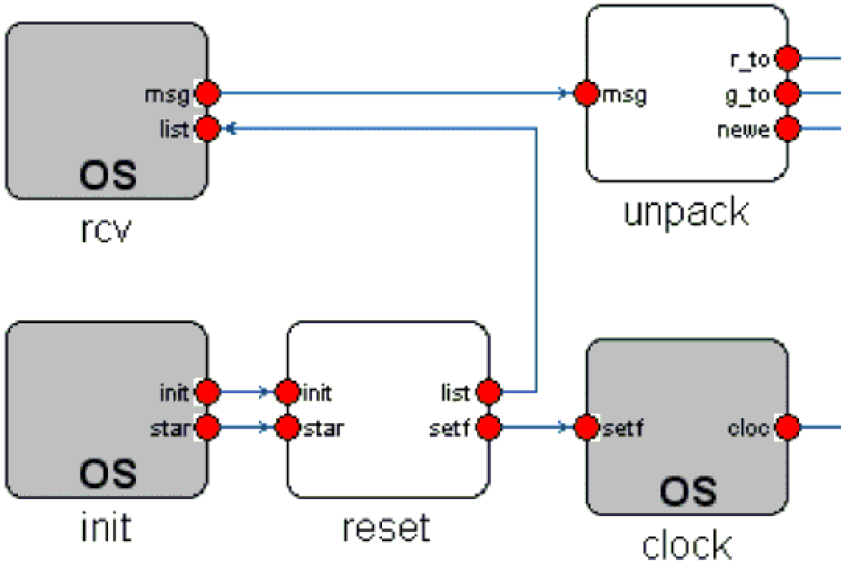
- Equipment and process control
  - Avionics
- Environment monitoring
  - Pollution
  - Chem/bio agent detection
- Target detection and classification
  - Acoustic and seismic beamformation
  - Target localization
- Smart structures
  - Acoustic sensing
  - Vibration control

NEST middleware

- Kind of distributed operating system that provides global services to the application
- The middleware must be
  - application specific
  - programming language and platform specific
  - highly configurable
- Currently, the middleware is written and verified for each application and platform

Our approach

- Automatically synthesize the middleware from abstract models
- Capture the temporal and computational aspects of distributed algorithms in a programming language and platform independent way
- Focus on composition and verification of middleware components



Asynchronous I/O automata

- Mathematical specifications of distributed algorithms
- Extensively used in the literature for the formal representation, verification and analysis of reactive systems [Lynch].
- Theoretical methods and results describing the interaction and refinement of asynchronous I/O automata.

The I/O automaton A

- $states(A)$, a nonempty set of states
- $start(A)$, a nonempty subset of $states(A)$
- $acts(A)$, a set of actions, partitioned into three sets: $in(A)$, $out(A)$ and $int(A)$, the set of input, output and internal actions
- $trans(A)$, a state-transition relation, where $trans(A) \subseteq states(A) \times acts(A) \times states(A)$
- $tasks(A)$, a partition of $out(A) \cup int(A)$

Execution of I/O automata

- The execution of A is a sequence $s_1, a_1, s_2, a_2, s_3, \ldots$ of states and actions such that s_1 is a start state, and (s_i, a_i, s_{i+1}) are transitions of A
- I/O automata are input enabled: input actions can occur in every state
- The execution order of locally controlled actions is nondeterministic

I/O automata specifications in practice

- States are described in terms of a list of state variables and their initial values, that is, $states(A) = D_1 \times \ldots \times D_n$
- Actions are grouped into logically coherent action groups, that is, $acts(A) = G_1 \cup \ldots \cup G_m$
- Each action group is parameterized, described in terms of action parameters: $G_i = E_{i,1} \times \ldots \times E_{i,p(i)}$
- Transitions are described in a mathematical pseudo-code accessing state variables and action parameters (preconditions and effects)

Focusing on structure

- The sets of states, actions and transitions are naturally structured in practice
- The pseudo-code describing the transition relation can be very complex and nondeterministic
- We minimize the complexity of the pseudo code by introducing more complicated compositional operators of I/O automata

Structured sets and data ports

- The following are structured sets
  - The domains of basic datatypes
  - Finite products of structured sets
  - Disjunct unions of structured sets
  - Finite powers of structured sets
  - The Kleene star of structured sets
- We can formally define data ports of structured sets that provide read and write access to limited parts of the structured set.

Structural model of I/O automata

- *states(A)* is a structured set
- *acts(A)* is a structured set
- *in(A)* and *out(A)* are data ports of *acts(A)*
- *value(A)*, a data port of *states(A)*
- *trans(A)*, only simple transitions are allowed:
  - Preconditions: only simple comparisons
  - Effects: only simple assignments

Compositional operators of structural I/O automata

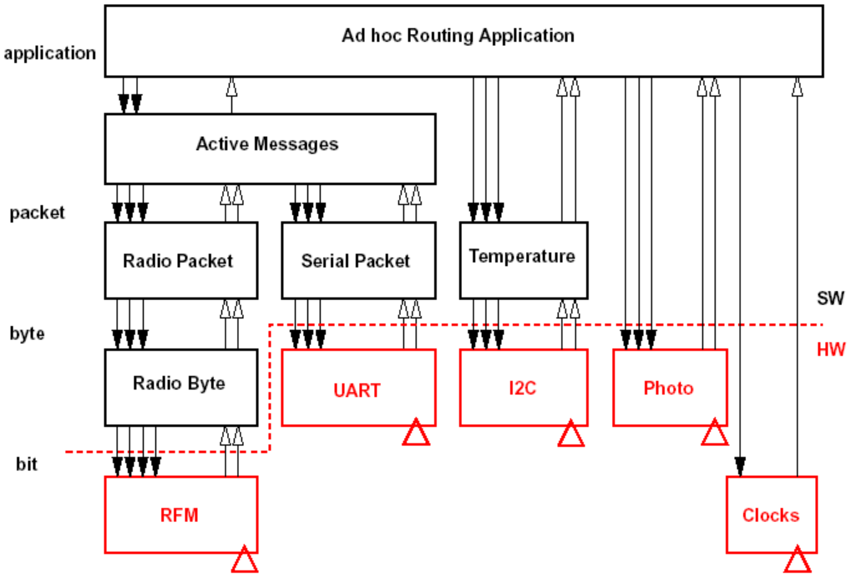
- **Variable:** most basic building block
- **Activator:** introduces new simple state-transitions
- **Product:** composition of automata
- **Union:** alternative implementations
- **Power:** implements arrays
- **Black Box:** wrappers around existing services

Case study: cooperative acoustic tracking

- Running on the Berkeley mote platform and the TinyOS operating system
- Two kinds of motes
  - Active tags: sends radio and sound signal
  - Trackers: receive radio and sound signal, compute time of flight of the sound, estimate distance
- Track table middleware service: maintain a table of all measurements at each tracker
- Based on the content of the track table, a local algorithm computes the location of the tags

TinyOS overview

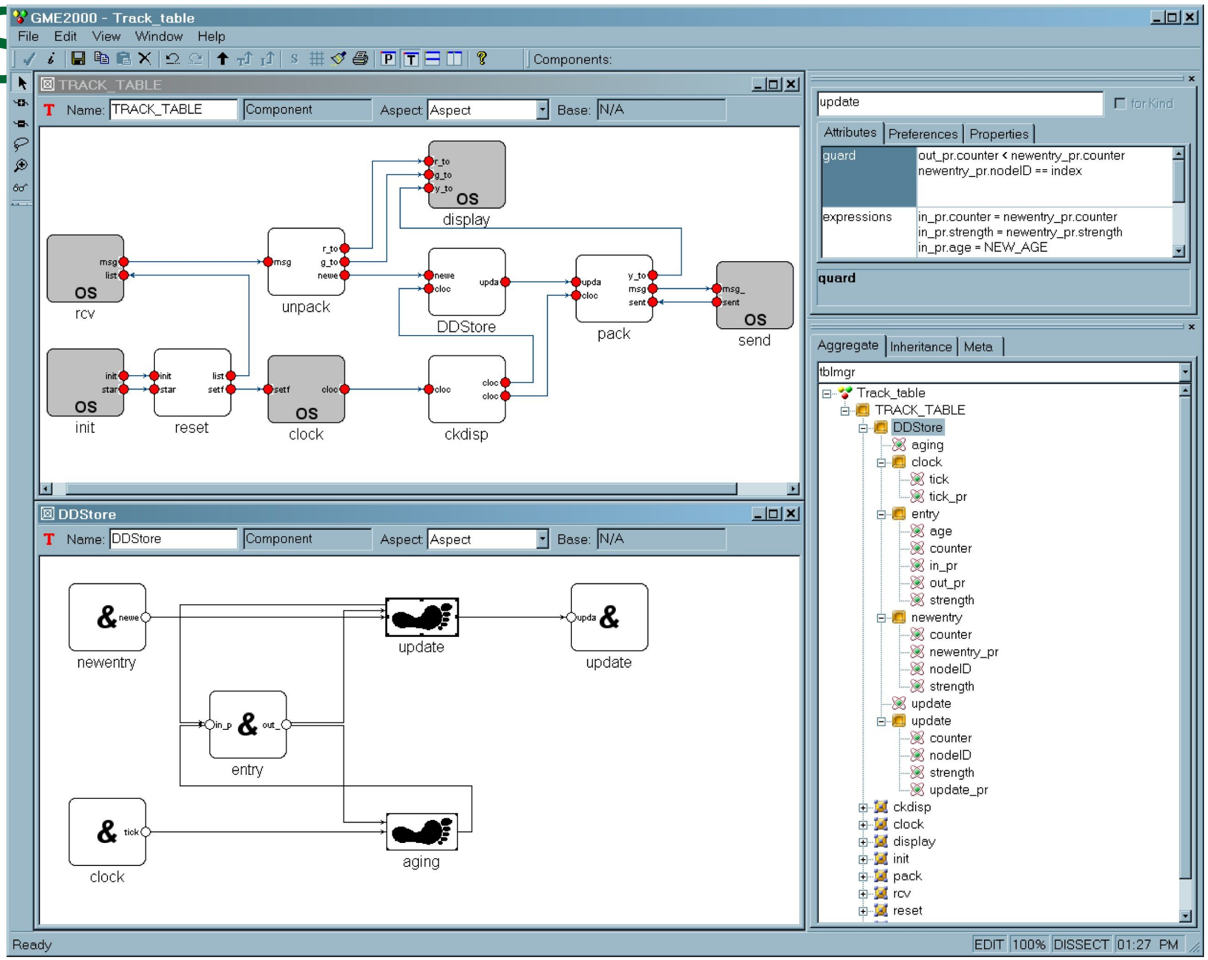
- The MICA and RENE hardware platforms (University of California, Berkeley)
- Event based operating environment for embedded networked sensors
- Two-level (non-preemptive) scheduling: events and tasks
- Components: event and command handlers, fixed memory frames
- The operating system / application is a collection of statically linked interacting components.



Generic Modeling Environment (GME)

- Configurable toolkit for creating domain-specific modeling and program synthesis environment
- Configuration is through a (UML based) metamodel specifying the syntactic, semantic and presentation information of the domain.
- A graphical model builder is used to build application models
- Domain specific synthesizers

Distributed Services Composition and Synthesis Technology (DISS



Case study: vibration control

- Acoustic damping of vibration in a fairing
- SIESTA: simulator implemented in Java that uses a simplified I/O automata-based middleware
- We used the same modeling environment (DISSECT) as before, but with a different conde synthesizer
- We modeled the broadcast and routing components

Conclusion

- It is possible to model distributed middleware services in a platform nutral and programming language independent way
- It is possible to automatically synthesize the distributed middleware from generic models using platform dependent code synthesizers
- Middleware service composition needs better support: a lot of research is still to be done