The Abstract Task Graph: Architecture-Independent Programming for Networked Sensor Systems

Amol Bakshi
Department of Electrical Engineering
University of Southern California
Los Angeles, CA USA 90089
amol@usc.edu

Joint work with: Viktor Prasanna (USC), Jim Reich and Dan Larner (PARC)
Outline

Background
Objectives and Key Concepts
Abstract Task Graph
  • Illustrative example
  • Syntax and semantics
The DART runtime system
Programming and Software Synthesis
Remarks
Networked Sensor Systems

Multi-tier, heterogeneous system architectures
Variety of sensing interfaces (locations, numbers, directionality)
Different computation, communication, and storage capabilities
Wireless micro-sensor networks for dense monitoring of physical environment
Context-aware spatial computing at all layers of the hierarchy
Broad interpretation of “sensor data”

What is Macroprogramming?

Specification of aggregate behavior in the sensor network

**Application-level** macroprogramming
- Define and **manipulate events at desired level of semantic abstraction**
- E.g.: ‘latest position of target’, ‘locations of all faulty sensors’, etc.

**Architecture-level** macroprogramming
- Concisely specify distributed computation and communication
- E.g.: ‘apply f(x) to all x within 10m’, ‘send data item d to 10 nearest nodes’, etc.

**Tradeoffs**
- Expressiveness
- Efficiency
- Reusability
- Automation
Two Approaches to System-level Support

**Application-specific**
- Define a list of programming language features for a specific application domain and implement the supporting protocols in the runtime system
- PROs: Efficient domain-specific language, reduced coding complexity
- CONs: Low code reusability across domains, difficult for application developer to add new features or modify existing features

**Application-neutral (ATaG)**
- Provide a small set of application-neutral primitives and focus on composability
- PROs: High reuse and extensibility, programmers can develop and add domain-specific libraries to the code base
- CONs: Application-oblivious runtime system might not be as efficient as application-specific implementations
Objectives of ATaG

Intuitive expression of reactive processing
Sense-and-respond application is a set of responses to a set of events with application-specific semantics

Spatial awareness and network awareness
Computing in a sensor network is centered around a real or virtual coordinate system (namespace) typically related to the physical environment

Architecture independence
Application development can proceed prior to decisions about the target deployment
Platform-specific and network-specific optimizations can be delegated to the compilation framework

Modularity and composability
Macroprograms will be ultimately generated from a higher level, purely declarative specification
Application behaviors as `services' which are composed at compile time or run time
Key Concepts of ATaG

Data-driven control flow
- An event can carry information about a phenomena or merely indicate its occurrence
- Task firing rules allow the specification of a range of execution patterns
- Scheduling and communication is managed by a runtime
- **Pros**: Modularity, reusability, extensibility, low program complexity
- **Cons**: Unpredictable delays, limited control over execution ordering

Mixed imperative-declarative program specification
- Separate “what” from “when and where”
- Imperative portion is a traditional sequential program (C/Java) using `get()` and `put()` for I/O
- Declarative portion can be specified visually
- Imperative and declarative parts can be modified independently
Example: Temperature Monitoring

Periodically collect and log temperature reading from all nodes at a designated root node
Periodically compare local reading at each node with readings at 1-hop neighbors
Signal an alarm if difference in reading between any pair of neighboring nodes is greater than a threshold
Syntax

An ATaG program is a set of “abstract” declarations

**Abstract task**
- corresponds to a type of processing
- has a unique name and associated user-supplied code
- task annotations determine placement and firing conditions

**Abstract data**
- denotes a type of application-specific data structure exchanged between instances of abstract tasks
- has a unique name and associated user-supplied data structure

**Abstract channel**
- associates abstract task with abstract data item; denotes I/O relationship
- specifies which instances of that abstract data item are of interest to the task
- annotations control the pattern of communication between task instances
Annotations: An Overview

ATaG Program

Abstract Task
- Firing rules ("when")
  - periodic
  - aperiodic
  - anydata
  - alldata

Abstract Data
- Placement ("where")
  - node ID(s)
  - geographic location(s)
  - resource
  - availability
  - degree of coverage

Abstract Channel
- neighborhood
  - distance
  - hops
  - k-nearest
- virtual topology
  - parent
  - children
- clustering
  - domain
  - instances-nearest
- initiation
  - push
  - pull
More Examples
Task scheduling
- Execution of an abstract task instance is atomic
- Task graph is executed in a breadth-first manner

Firing rules
- **Periodic**: Task instance is scheduled when periodic timer expires
- **Any-data**: Task instance is scheduled when an instance of any of its input data items is available
- **All-data**: Task instance is scheduled when an instance of each of its input data items is available

The data pool
- Global communication buffer managed by the runtime
- Policies for managing the data pool depend on capabilities of target platform (e.g., buffered vs. un-buffered, static vs. dynamic memory allocation)
get() and put()

Communication orthogonality
- property of ‘generative communication’ in tuple spaces
- sender and receiver are not aware of each other
- data sharing is decoupled in space and time

Network stack managed entirely by runtime
- Application-level tasks only use get() and put()
- All send()/receive() invocations at network level are implicit in the annotated get()/put() invocation
- Hides heterogeneous and/or dynamic nature of distributed communication
- Allows ‘low-level’ optimizations to be implemented in the runtime and hidden from the programmer

Simple to understand for “non-expert”
Programming and Software Synthesis

Target network description
[Annotated Network Graph]

Application specification
[Abstract Task Graph]

COMPILE
Analyze ATaG and ANG

Generate configuration files

Synthesize compile-ready code

Generate task code template

Application-specific task code

Runtime system template

USER INPUT

June 24, 2005 CENS Technical Seminar Series Amol, 14/20
Step 1: Visual Programming (Declarative)
Step 2: Populating code skeleton (imperative)

```java
public class SampleAndThreshold implements Runnable {
    private int latestReading;
    private static int oldReading;
    private static boolean acquired=false;

    public SampleAndThreshold(DataPool dp, Config myconfig,
                               NetworkArchitecture t_networkArchitecture, mGUI t_GUI)
    {
        m_aSensor = new Sensor(myconfig.myID(), Constants.ACOUSTIC_SENSOR);
        [...]
    }

    public void run() {
        try {
            for (; ; ) {
                latestReading = m_aSensor.reading();
                m_dataitem = new DataItem(IDConstants.D_TARGETALERT,
                                           IDConstants.T_SAMPLEANDTHRESHOLD, m_targetAlert);
                if (latestReading > 0) {
                    m_targetAlert.setDistance(latestReading);
                    m_targetAlert.setAcquired(true);
                    if (!acquired)
                        acquired = true;
                    m_dataPool.putData(m_dataitem);
                } else if (latestReading == 0 && oldReading != 0) {
                    acquired = false;
                    m_targetAlert.setAcquired(false);
                    m_dataPool.putData(m_dataitem);
                }
                oldReading = latestReading;
                Thread.sleep(3000);
            }
        } [...]
    }
}
```

Maintain local state in static variables

Create notification of acquisition or loss of target
DART: Data-driven ATaG RunTime

Responsibilities
Support distributed data-driven execution
Interpret task and channel annotations at run time
Provide spatial awareness and network awareness

Approach
All nodes run essentially the same DART engine
Appropriate user-level code is provided for each assigned task
Compiler generates app-specific glue code and a config file for each node

- **Sensors**
  - UserTask1
  - UserTask2
  - ... (...)

- **Actuators**

- **NetworkArchitecture**
  - Maintain neighborhood information, virtual topologies, logical namespace, etc.

- **NetworkStack**
  - Routing, medium access, physical layer protocols

- **DataPool**
  - getData() and putData()
  - concurrent access, reference counts

- **DISCh**
  - Translate annotations, dispatch data to other nodes

- **ATaGManager**
  - Store task code, task/channel annotations, spawn tasks
Customizing DART

ATaGManager – tasks and channel annotations
Data structures for data pool manager

NetworkArchitecture
• Extent of neighborhood (distance and hops)
• Virtual topology protocols (if any)

Per-node configuration
• Depends on target platform
• Node ID, location (if known a priori), etc.

Lines of code (Java)
• DART: ~2800 (single-machine simulation)
• Application-level tasks (object track): ~100
• Glue code (object track): ~15
• Config files: $n$ lines for $n$-node network
Related Work

Parallel and distributed computing
- Blackboard architectures: AI research in 1970s, DOSBART [Larner '90]
- Tuple spaces: Linda [Gelernter '85], LIME [Picco et al, '99], TinyLIME [Curino et al, '05]
- Dataflow synchronization: Distributed Oz [Haridi et al, '98], Data Driven Graph [Tran et al, '99]

Sensor networking
- Functional programming: Regiment [Newton, Welsh '04]
- Centralized memory/CPU model: Kairos [Gummadi et al, '05]
- Logic programming: Semantic Streams [Whitehouse et al, '05]
- Other: TinyDB, State-centric programming, ...
Remarks

Data driven program flow is an attractive paradigm to express reactive processing and enables distributed sharing.

Mixed imperative-declarative programming is the key to architecture independence.

Our programming model hides the network architecture; DART hides the platform architecture.

ATaG programming only requires knowledge of a traditional programming language such as C or Java.

Future work
- Managing sensing resources in DART
- Dynamic task instantiation
- Performance-related annotations