Tilting at Windmills

FULL PRESENTATION

Jan Vitek Suresh Jagannathan
Purdue University

1. Introduction

Harnessing wind energy is one of the pressing challenges of our time. The scale, complexity, and robustness of wind power systems present compelling cyber-physical system design issues. In response to these emerging challenges, Purdue has initiated strategic partnerships for infrastructure development. Working with the US Department of Energy, Purdue is developing physical infrastructure for an experimental wind farm on campus. Separately, General Electric has committed to building 60 wind turbines with total capacity exceeding 100 MW on land owned by Purdue. In return, GE will allow three turbines to be utilized for research and education, along with access to unrestricted data from all other turbines. This represents the most comprehensive experimental and operational wind energy infrastructure available for academic research today.

Figure 1: **(a)** *The wake field of a wind farm decreases performance of down-wind turbines, and* **(b)** *turbine failure caused by a combination of high winds and control system failure.*

Leveraging this physical infrastructure, we aim to develop a comprehensive computational infrastructure for programming distributed real-time embedded devices that collaborate to perform complex, time critical tasks amenable to formal verification. In contrast to traditional efforts that focus on programming-in-the-small, we emphasize *declarative specifications, robustness, longevity, and assurance*. We motivate our ideas with two examples. Fig. 1(a) illustrates the wake field of an operational wind farm. While the first (upwind) set of turbines encounter clean air, downwind turbines are subject to considerable turbulence, and associated loss in performance. Studies have hypothesized that this loss can be as high as 25% [3]. Integrated control of the farm, where each turbine is cognizant of the prevailing wind direction and rotor loading state as well as the down-wind turbines and their loading states, can significantly enhance the performance of the entire farm. Even a 1% productivity increase from a 1000MW farm is enough to power 2,500 homes. The implications of integrated control on longevity of turbine components and cost of energy are even more significant over the 30 year life of the farm. Beyond these, there are more subtle benefits from an integrated approach to wind-farm control. These include noise control during quiet hours – directly impacting setbacks (EPA/ local regulations), installation density, and consequently cost.

To address these challenges we have embarked on the construction of ENSEMBLES, a computational infrastructure for coordinating complex behavior in heterogeneous distributed real-time embedded (DRE) systems, and using the infrastructure to develop an automated wind farm control and sensing software system. A key component of this infrastructure is a programming model that includes mechanisms for declaration of real-time and macroprogrammed constraints and performance requirements, mechanisms for distributed coordination among single-node virtual machines, and support for highly heterogeneous compute environments encountered in our testbed. Our distributed services-layer includes a macroprogramming language for specification of coordination mechanisms, with support for control feedback primitives, and computational optimizations, including load conditioning, network services, and service classes. A complementary specification language based on temporal logic is used to verify correctness constraints and to provide a high-degree of assurance for deployed applications.

2. The ENSEMBLES Cyber Infrastructure

Our design focuses on *programmability* and *assurance* for DRE systems. Technically, our approach rests on two principles: *macroprogramming* and *virtualization*. Macroprogramming is a programming methodology that allows users to express global system behavior through high-level declarative specifications, which are then translated to consistent node-local behavior. Virtualization provides a uniform interface to a heterogeneous environment. In our case, we rely on a high-level language virtual machine coupled with an abstract API for various sensors and actuators on the wind turbine. A wind turbine data acquisition and analysis system (WT-DAAS) is built on top of this infrastructure that is used to program health monitoring and optimization algorithms that address the *performance* and *fault tolerance* challenges of wind engineering. There are two key technologies involved:

- **mPL:** The macroprogramming layer [1, 2], is based on mPL, a new macroprogramming language that lets users specify coordinated behavior, as well as to set optimization targets for resource usage. With mPL, the behavior of the entire farm, as well as individual turbines, can be programmed declaratively. A high-level specification language based on temporal logic is compiled to lower-level mPL code along with safety and liveness proofs derived from the logic.
- **mOS:** Virtualization is achieved by mOS, a runtime system that is implemented on top of Ovm, our Safety Critical Java virtual machine [6]. Node-level behavior can thus be coded in Java which provide software engineering and assurance benefits over low-level languages such as C. mOS is a customizable node operating system that can run on native hardware on small devices, as well as on top of a real-time operating system when available.

The notion of an *ensemble* is introduced by mPL to group interconnected computational elements (sensors, actuators, controllers), some of which can be ensembles themselves. Thus, one can view an entire farm as an ensemble of turbines, with each individual turbine in turn as an ensemble of sensors, actuators and compute nodes. mPL provides the means to define behavior over ensembles and declaratively control low-level details such as power consumption, resource management, thread scheduling, and sensing. mPL makes both distribution and replication-based fault-tolerance transparent. The mOS layer relies on Java for portability, concurrency, real-time primitives, and dynamic loading. The last feature is key to supporting software updates to a running wind farm. The particular subset of Java we have adopted for mOS is Safety Critical Java [6], a variant that is being designed by the PIs for certification. The mOS layer supports software assurance by enforcing strong isolation between components and generates trace information for off-line analysis in case of failures.

Macroprograms specify distributed behavior based on dataflow abstractions. mPL views computations as networks of *functional components* or FCs communicating synchronously through channels. Macroprograms further specify constraints on the computation. It is these constraints that guide the deployment of the macroprogram on the individual nodes. The design of mPL is motivated by the following design choices:

- **Composition and validation:** A macroprogram is composed of reusable components with typed, verifiable interfaces. High level abstractions are implemented as compositions of FCs. These FCs make low-level details transparent to the application.
- **Dataflow:** Communication, both local and over the network, between FCs is transparently supported as a primitive. Dataflow is mostly synchronous, components are triggered in response to inputs. Real-time considerations usually require that components operate at periodic intervals.
- **Heterogeneity:** mPL allows effective utilization of heterogeneous nodes in the network, based on their capabilities (e.g., if a node has a sensor, or has large memory). This provides a unified programming model that is aware of hardware capabilities.

An mPL macroprogram provides a distributed behavioral specification of how an application can be instantiated on physical nodes in the network. The specification consists of FCs and an *interaction assignment* (IA), which is a directed graph that specifies dataflow through FCs. Each FC has typed data inputs and outputs, and represents a specific data processing function. At design time, a specification of each FC is provided; for the macroprogram developer, it is only this specification that is relevant. Associated with each FC is a handler that implements the specification. For example, the following defines a FFT component:

```
fft : {  mcap = MCAP_FAST_CPU, fcid = FCID_FFT,
         in[craw_t ], out[freq_t] }
```

This asserts that the FFT handler can only be executed on machines with fast CPUs, that it must have type FCID_FFT, and takes as input a compressed raw data stream and outputs frequency domain data. FCs are interconnected via synchronous data channels; asynchronous communication can be used at the risk of not being able to validate timing requirements. An output of an FC may be connected to the input of another FC, in an IA, only if their types match. Thus, an IA can be statically type-checked. In addition to dataflow, an IA also specifies data producers and consumers, which correspond to source and sink *device ports*, respectively. Device ports are logical and may not always correspond to hardware devices.

Our aim is to have mPL programs easily and directly specify distributed system behavior using simple declarative syntax. Composing applications with reusable components allows the macroprogrammer to focus on application specification rather than low-level details or inter-node messaging. Furthermore, high-level abstractions can be used to refine and tune the semantics of the applications (some of which the reader might have noticed are omitted in our example code) and provide rich features such as neighborhood communication and load conditioning.

The mPL compiler reads an mPL program and generates an annotated directed graph representation. To execute an application, this graph is communicated over the network nodes running mOS. FCs are placed in the program memory of the nodes either when mOS is being flashed, or by downloading them over-the-air. The node capability constraint determines if a node receives a particular FC or not. Each node in the network receives a subgraph of the IA, consisting only of FCs that can be instantiated at the node. Dataflow with remaining FCs is transparently handled through a network and routing layer implemented as part of mOS. Local dataflow is established by the mOS runtime system. mPL supports hierarchical aggregation of FCs into ensembles, which can be viewed as FCs themselves and further composed.

3. Conclusions

Our undertaking seeks to apply programming language concepts and runtime system support for distributed embedded systems, to improve fault tolerance, reliability, and real-time support in heterogeneous environments. These developments are motivated by, and contribute to problem solving in design and optimization of large-scale wind-farms. These problems include physical models underlying coordinated design and optimization, sensing, model reduction, and control for real-time performance, safety and longevity, implementation on large-scale distributed environments, and validation on experimental and operational testbeds.

Acknowledgements. Support for this work is provided by the National Science Foundation under grant **CNS 1136045**.

References

[1] Asad Awan, Suresh Jagannathan, and Ananth Grama. Macroprogramming heterogeneous sensor networks using cosmos. In *EUROSYS*, 2007.

[2] Asad Awan, Ahmed H. Sameh, Suresh Jagannathan, and Ananth Grama. Building verifiable sensing applications through temporal logic specification. In *International Conference on Computational Science (1)*, pages 1205–1212, 2007.

[3] R. Barthelmie and L. Jensen. Evaluation of wind farm efficiency and wind turbine wakes at the nysted offshore wind farm. volume 13, page 573âĂŞ586, 2010.

[4] David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proc. of PLDI '03*, June 2003.

[5] Chih-Chieh Han, Ram Kumar Rengaswamy, Roy Shea, Eddie Kohler, and Mani Srivastava. SOS: a dynamic operating system for sensor networks. In *Proc. of MobiSys '05*, June 2005.

[6] Thomas Henties, James Hunt, Doug Locke, Kelvin Nilsen, Martin Schoeberl, and Jan Vitek. Java for safety-critical applications. In *Certification of Safety-Critical Software Controlled Systems (SafeCert)*, March 2009.

[7] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proc. of ASPLOS-IX*, November 2000.

[8] Hongzhou Liu, Tom Roeder, Kevin Walsh, Rimon Barr, and Emin G. Sirer. Design and implementation of a single system image operating system for ad hoc networks. In *Proc. of MobiSys '05*, June 2005.

[9] Ryan Newton and Matt Welsh. Region Streams: functional macroprogramming for sensor networks. In *Proc. of DMSN '04*, August 2004.

[10] Ramkumar Rengaswamy, Eddie Kohler, and Mani B Srivastava. Software based memory protection in sensor nodes. In *Proc. of EmNets'06*, May 2006.