PROFILING PRIMITIVES OF NETWORKED EMBEDDED AUTOMATION

A Thesis

Presented to

The Graduate Faculty of The University of Akron

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

Branden Archer

May, 2010

PROFILING PRIMITIVES OF NETWORKED EMBEDDED AUTOMATION

Branden Archer

Thesis

Approved:

Accepted:

Advisor
Dr. Shivakumar Sastry

Department Chair
Dr. Alex De Abreu Garcia

Committee Member
Dr. Nathan Ida

Dean of the College
Dr. George K. Haritos

Committee Member
Dr. Igor Tsukerman

Dean of the Graduate School
Dr. George R. Newkome

Date

ABSTRACT

Recent advances in wireless technologies now allow microcontrollers with low-power wireless radios to integrate with sensor and actuator systems. Using such devices to perform automation tasks is referred to as Networked Embedded Automation (NEA). In NEA, devices perform local actions based on local sensing and communications with neighboring devices. Traditional automation systems use networks of Programmable Automation Controllers wired to each other to regulate system behavior. NEA represents a change in direction for automation systems as they are poised to leverage existing wireless sensor network technologies and distributed algorithm advances.

A challenge with integrating NEA into automation systems is understanding predictability and safety issues that are inherent in these enhanced systems. This thesis proposes a set of primitives that are suitable for developing composable automation systems that are regulated by embedded devices interacting over wireless links. A node level architecture and design that helps realize these primitives is presented. Using a robust platform called FireFly, the temporal performance of these primitives is profiled. Experimental results demonstrate that the latency and jitter associated with these primitives are promising and useful for a class of automation

systems. In the future, this work can be extended to larger scale systems with tighter temporal constraints.

ACKNOWLEDGEMENTS

First of all, I would like to thank my adviser, Dr. Shivakumar Sastry, for his support and guidance that made this thesis possible. Thanks also goes to my committee members, Dr. Nathan Ida and Dr. Igor Tsukerman, for their comments and feedback on this thesis. I want to express my thanks to Gay Boden for helping with all the small issues that tend to come up and generally making life easier. My gratitude also goes to my laboratory colleagues, Kranthi Mamidisetty, John McGonnell, Mukesh Chippa, and Chidi Anozie, for their support. A big thank you especially to John, for without John's interface board designs and help in fabrication the demonstration of this thesis on the model factory would not have been possible.

I also thank my mentors from my undergraduate co-op, Lincoln Dickerson and Jeff Spencer. Without their lessons in proper software development this thesis (and much of my other research) would be *much* more difficult and not nearly as elegant.

Finally, I dedicate this thesis to my mother who has kept me going with her love, support, and delicious cooking!

TABLE OF CONTENTS

	Page
LIST OF TABLES	viii
LIST OF FIGURES	ix
CHAPTER	
I. INTRODUCTION	1
1.1 Contributions	4
1.2 Overview of this Thesis	4
II. BACKGROUND	6
2.1 Microcontroller Platforms	6
2.2 Networked Embedded Automation	7
2.3 PLC Scan Architecture	8
2.4 Multiaccess Communication Protocols	9
III. TDMA COMMUNICATION PROTOCOL	13
3.1 Design Objectives	13
3.2 Design of the TDMA protocol	15
3.3 Implementation	17
IV. BEHAVIOR SPECIFICATION AND ITS EXECUTION	23

4.1	User Level Behavior Specification	23
4.2	Device Level Behavior Specification	28
4.3	Device Specification Evaluator	31
V.	NOTE LEVEL ARCHITECTURE	33
5.1	Specification Program Input	33
5.2	Specification Memory Storage	34
5.3	Mapping Execution to the Environment	36
5.4	Specification Execution	37
VI.	SYSTEM LEVEL CONSIDERATIONS	39
6.1	Periodic Scan Time	40
6.2	Local Response	40
6.3	Remote Response	41
6.4	Request/Response	42
6.5	Publish/Subscribe	43
6.6	Multihop Publish/Subscribe	44
6.7	Multihop Request/Response	47
VII.	IMPLEMENTATION RESULTS	49
7.1	Performance Evaluation	49
7.2	Model Factory Deployment	57
VIII.	CONCLUSION	60
	BIBLIOGRAPHY	62

LIST OF TABLES

Table	Page
7.1 Notation	51

LIST OF FIGURES

Figure	Page
2.1 Scan architecture of PLCs	8
3.1 TDMA frame	16
3.2 Format of a packet. Numbers represent size in bytes, where n is variable length	18
3.3 Example timing layout of a slot, times not to scale	20
3.4 Successful packet transmission rates based on a 26 byte packet in a 2ms slot	21
4.1 A microcontroller regulates local operations and interacts with its upstream and downstream units via communication channels InCh and OutCh, respectively	25
4.2 Example user level behavior specification for a conveyor unit	26
4.3 Example device level behavior specification for a conveyor unit; the $\leftarrow\perp$ symbol means the line continues unbroken	30
4.4 Extending the scan architecture of PLCs	31
5.1 Definition of a variable	35
5.2 Definition of a token	36
6.1 Periodic scan time primitive example	40
6.2 Local response primitive example	41

6.3	Remote response primitive example	42
6.4	Request/Response primitive example	43
6.5	Publish/Subscribe primitive example	45
6.6	Multihop publish/subscribe primitive example	46
6.7	Multihop request/response primitive example	48
7.1	Execution time of evaluation engine with no sensor inputs, actua- tions, or communication	50
7.2	Execution time of evaluation engine with no sensor inputs or actu- ations, with background communication	51
7.3	Response time of evaluation engine with sensor input and actuation, but no communication	52
7.4	Response time of evaluation engine with sensor inputs and back- ground communication	53
7.5	Response time of 1-hop 1-way communication from a sensor to an actuator over a wireless link	54
7.6	Response time of a request/response primitive with 2-way commu- nication over a 1-hop network	55
7.7	Actuation response times between two nodes under variable load	56
7.8	Multihop Publish/Subscribe scalability	57
7.9	Multihop Request/Response scalability	58
7.10	Drawing of the microfactory	59

CHAPTER I

INTRODUCTION

Advances in low-power wireless communication technology and embedded microcontrollers have made feasible a class of systems called *Networked Embedded Automation* [1]. In such systems, deeply embedded microcontrollers, each connected to local sensors and/or actuators, coordinate local operations over low power wireless links to regulate system-level behaviors of automation systems. Integrating such devices with existing automation systems will allow finer-grained control, reconfigurable topologies, improved diagnostics, and reduced system lifecycle costs. Because such systems represent a significant departure from the current state-of-practice in a critical application domain, it is important to gain insight into the factors that impact their design and performance.

Automation systems today represent a key part of our world economy. In 2008 the top 50 automation companies grossed over \$87 billion in worldwide sales [2]. These systems are used to automate three types of tasks: 1) periodic and repetitious tasks, 2) tasks which may be hazardous for a human to perform, and 3) tasks which require precision and predictability better found when using robotics. Many different industries use automation technologies, including manufacturing, agriculture, and aerospace. The current state of the art in these systems is the Programmable Au-

tomation Controller (PAC), which supports the regulation of devices through wired sensors and actuators, and supports communication with other PACs through specialized wired networks such as EtherNet/IP and ControlNet [3].

Networked embedded automation represents a change in direction from current automation systems. Today's automation installations usually consist of several hierarchically organized PACs. These installations are reliable and predictable. However they are also brittle, in that only well-known faults can be detected and changes to the system require much effort. Further, they do not leverage current distributed system advances; communications are limited to mode management, transfer of control, and status reporting. Networked embedded automation is poised to take advantage of wireless sensor network technologies and distributed algorithm advances, but their adoption presents major challenges to existing automation systems. It is necessary to assure designers that by moving to new technologies that the reliability, predictability, and safety of the integrated system will not be compromised.

Among these challenges, this thesis addresses predictability in the context of networked embedded automation. It is critical to understand how predictability can be achieved at the node level and at the level of communication among nodes. To achieve predictability, an efficient runtime engine was developed and integrated with a time-triggered communication backbone for individual nodes in a networked embedded automation system. This runtime is an expression evaluator similar in behavior to the Ladder logic language used in PACs. Predictability is characterized in terms of latency and jitter for node level execution and inter-node communications. To achieve

predictability at the node level a scheme was designed that periodically gathers data from local sensors and a received-communication buffer, evaluates expressions by locking the input data, actuates local devices, and fills a send-communication buffer. Predictable inter-node communications is achieved by using a statically scheduled TDMA protocol where wireless transmissions between nodes occur in well defined, collision free time slots. More about the design of the expression evaluator and TDMA protocol is in Chapters 5 and 3 respectively.

Evaluating the predictability of the proposed scheme cannot be done in isolation, as a context is necessary to understand typical operating scenarios. To this end, a collection of behavioral primitives were developed to provide a holistic operating context. After implementing the proposed scheme on the Firefly platform, the performance of these primitives was analyzed. The results show that even with the current level of reliability in low power wireless links, networked embedded automation is viable. To demonstrate this viability, a collection of Firefly nodes was deployed to regulate the behavior of a microfactory available at the Complex Engineered Systems Laboratory at the University of Akron. More about the proposed primitives is discussed in Chapter 6 and their performance on the Firefly platform is discussed in Chapter 7.

1.1 Contributions

In summary, the contributions of this thesis are:

1. Design and implementation of a predictable statically scheduled TDMA communication protocol on the Firefly wireless sensor platform.
2. Design of a behavioral specification for discrete networked embedded automation.
3. Design and implementation of a memory compact expression evaluator runtime for the behavioral specification on the Firefly wireless sensor platform.
4. Identification and evaluation of a set of behavioral primitives used as building blocks for achieving predictability.
5. Demonstrate how the identified primitives map to realistic systems by deploying Firefly nodes executing the expression evaluator runtime to regulate the behavior of a model microfactory.

1.2 Overview of this Thesis

Following some relevant background in Chapter 2, Chapter 3 shows the design of the TDMA scheme. Chapter 4 presents the behavioral specification used and a design for an expression evaluator runtime. Chapter 5 then presents details on the implementation architecture for the expression evaluator runtime on the Firefly platform. The

networked embedded automation primitives are presented in Chapter 6, followed by evaluation results in Chapter 7. Finally, the conclusion is in Chapter 8.

CHAPTER II

BACKGROUND

2.1 Microcontroller Platforms

Several node platforms, such as the Mica2, MicaZ, Telos, TinyNode, and Firefly, are used in sensor networks. The first four of these typically execute a reconfigurable, component-based, event-triggered operating system and application framework called TinyOS [4]. While this approach is flexible, timing predictability was not its primary design objective. The real-time operating system (RTOS) Nano-RK [5] is available for the Firefly and MicaZ platforms.

The first two platforms, Mica2 and MicaZ [6], use the Atmel ATMega128L microprocessor with 4KB of RAM and 128KB of ROM. The Telos [7] and TinyNode [8] platforms both use a Texas Instruments MSP430 microprocessor with 10KB RAM, 48KB ROM and 8KB RAM, 92KB ROM respectively. The Firefly [9] uses an Atmel ATMega1281 microprocessor with 8KB RAM and 128KB ROM, and has an available mini-SD card slot. The Mica2 uses the CC1000 radio which operates on the 315, 433, 868, and 915 MHz bands, and uses no packet buffer for transmitting or receiving [10]. The MicaZ, Firefly, and Telos all use the Chipcon CC2420 IEEE 802.15.4 standard-compliant radio transceiver with a 128 byte packet buffer and maximum data rate

of 250Kbps [11]. The radio used on the TinyNode is the Semtech SX1211 which operates in the 863-870 MHz, 902-928 MHz and 950-960 MHz ranges, has a packet buffer of 64 bytes, and has a maximum data rate of 200Kbps [12].

2.2 Networked Embedded Automation

Automation systems are distributed real-time systems that regulate the coordinated operations of humans and machines in a safe, predictable, and reliable manner. These systems use thousands of sensors to detect phenomena in the physical world and regulate operations via actuators. The logic for computing values of actuators executes on specialized industrial controllers.

Networked embedded automation offers significant opportunities such as dynamically configurable system topologies, readily deployable wireless sensor nodes, and improved capabilities for safety, monitoring and diagnostics. For example, a collection of wireless sensors used to augment an existing automation system can detect deviations from the expected conditions as soon as possible, and perhaps prevent unnecessary operations on a defective part or deflect the defective parts to a repair or recycle stream of flow. In applications such as drilling rigs, early monitoring of operational conditions can significantly reduce the cost of operating and maintaining the system.

Figure 2.1: Scan architecture of PLCs

2.3 PLC Scan Architecture

To achieve predictable operations, most industrial controllers use a periodic scan architecture, depicted in Figure 2.1. At the beginning of the scan, the environment is sensed and copied into an input image table to ensure that the values do not change during evaluation. Next, output values are computed and held in the output image table until the end of the evaluation and presented. After the evaluation is complete the output image table is copied to local actuators. These actions repeat periodically. Before each evaluation of the logic, values of the output image table are restored to their default state. Chapter 4 builds on this periodic scan architecture to develop predictable node level behavior.

2.4 Multiaccess Communication Protocols

Wireless communications in wireless sensor networks can use either a CSMA (Carrier Sense Multiple Access) or a TDMA (Time Division Multiple Access) technique to provide access to the medium. CSMA-based techniques are good for highly dynamic environments when throughput requirements are low and when power consumption is not a dominant constraint. This is because (a) any packet collisions will lead to lost packets and cause packet retransmissions, and (b) nodes must periodically sample the medium to check for intended messages whether they are present or not. In contrast, TDMA-based techniques offer predictable end-to-end timing performance, higher throughput if necessary and better energy performance. For this investigation TDMA based techniques are a better fit. Several different TDMA protocols have been proposed in the literature and can be classified as either loosely synchronized, where the nodes do not share a synchronized time base, or tightly synchronized, where nodes are time synchronized. A few protocols from each classification are discussed below.

2.4.1 Loosely Synchronized Protocols

All of the loosely synchronized protocols mentioned use a form of CSMA for packet transmission, but use some method for reducing collisions. S-MAC [13] and T-MAC [14] use an idea of *virtual clustering* to pair neighboring nodes to coordinate active and sleep schedules. Communications occur during the active mode. Coordination is accomplished with periodic synchronization messages between neighbors.

The difference between these two MACs is T-MAC will reduce its active mode to save energy by bursting its messages at the start of the mode and time-out at the end. Both of these MACs favor energy conservation over node fairness and reduced latency.

Z-MAC [15] is a hybrid protocol that adapts based on the level of congestion. Under low congestion it acts like a CSMA protocol, and under high congestion it acts like a TDMA protocol. Time is divided into 50ms slots. At system activation nodes discover their neighbors and determine their slots. This MAC allows for the recapture of wasted slots without renegotiating the slot schedule, which disallows collision-free transfer of messages.

The DESYNC [16] protocol is based on desynchronization, where nodes try to space their communications as far away from other nodes as possible. By doing so, slot sizes are automatically regulated and slots are appropriately spaced. The scheme is decentralized, simple, and adaptive to adding and removing nodes from the system.

There are two issues with these protocols with respect predictability. First, communications are not guaranteed to be collision free. Second, these protocols have a high and unpredictable CPU overhead. This overhead is due to either processing messages to maintain message slots (especially in the case of DESYNC) or receiving a variable number of messages in a given time window, which is only bounded by the maximum number of messages possible within the window. CPU overhead prevents an underlying application from executing. The first issue is solved by the tightly

synchronized protocols in the next section, while the second issue is solved by only one of the presented protocols.

2.4.2 Tightly Synchronized Protocols

TRAMA [17] uses a time synchronized frame of slots consisting of both scheduled and random access slots. Scheduled slots are collision-free whereas the random access slots use CSMA. A distributed election scheme based on traffic patterns at each node determines which nodes transmit in which scheduled slot. Random access slots are used for network management and node admission.

LMAC [18] uses a distributed algorithm to assign nodes to a single slot in a time synchronized frame. Nodes determine their slot assignment based on local information. A slot consists of two sections, a control message and a data unit. An owner of a slot will always transmits. The control message declares to which node the data unit is sent. If a node is addressed in the control message it will listen for the data unit; otherwise it will sleep until the next slot.

There are several issues with the above mentioned protocols which prevent a simple and predictable implementation. First, for both TRAMA and LMAC, time synchronization is required for correct operation but is considered as a separate issues from the protocol. Further, in both of these protocols the maximum number of packets a node may receive in a time window is not reasonably bounded. In the case of TRAMA, a node will know ahead of time which slots contain packets destined for it. However, for both protocols the maximum number of packets a node can receive

in a frame is the maximum communication slots in the frame. This can result in significant processing overhead which may vary from frame to frame.

RT-LINK [19] has a novel approach to solving the burden of synchronization; it achieves synchronization either through special hardware listening to periodic out-of-band pulses from an AM signal broadcaster, or through in-band software-based synchronization messages from a dedicated node. A frame is sectioned into 32 slots, containing both scheduled and random access slots. Scheduled slots are collision-free, and random access slots employ slotted-ALOHA. For each scheduled slot a node knows if it could receive a packet during the slot or not. This gives a better bound to the maximum number of packets that can be received during a frame, as many slots can simply be ignored. Each slot is 8 ms, giving a 256 ms frame. Nodes may use one or more slots per frame, but the slot schedule is statically assigned. A rate limiting settings is available for scheduled slots, where a node can communicate during every frame, every other frame, etc. A protocol similar in spirit to RT-Link is designed and implemented for this investigation and details are provided in Chapter 3.

CHAPTER III

TDMA COMMUNICATION PROTOCOL

This chapter presents a design of a in-band time-synchronized TDMA scheme which allows for collision-free communication by using statically scheduled slots. The design was implemented on the Firefly wireless platform which uses the CC2420 radio coupled with an ATmega1281 8-bit microcontroller with 8KB of RAM and 128KB of ROM. Following the design, implementation results demonstrating communication success rates are shown.

3.1 Design Objectives

The purpose of using a TDMA communication protocol is to make communication latency and jitter predictable. One unavoidable problem that arises when implementing a TDMA scheme is time synchronizing all nodes to facilitate communication within well defined time slots. There are several ways to accomplish this, including executing a time synchronization protocol on all nodes, or using special hardware to facilitate out-of-band synchronization using an external time base. Adding additional software for a time synchronization protocol may decrease the timing predictability of the system, and requiring special hardware will increase the cost of the system,

both of which should be avoided. For these reasons, a simple time synchronization mechanism needs to be used.

Beyond synchronization, the resources available on the target platform must be considered. A Firefly node contains a restrictive amount of memory. The TDMA protocol must require little memory. This restricts the flexibility of the protocol as send and receive buffer space is limited. Additionally, the CC2420 radio is only capable of transmitting at a raw speed of 250Kbps. The overhead of detecting message reception failures and resending is an expensive luxury.

One final point to consider is the duration of a communication slot. The CC2420 radio contains a 128 byte send and receive buffer. The radio transmission time for the largest possible packet is around 4ms. In practice, however, this time is larger when including SPI transfer times to and from the radio, software overheads such as checksum computations, and guard times used to compensate for processor clock drift. Having a small slot time is desirable, as it equates to quicker response times for applications.

For these reasons, the main design objectives for the TDMA protocol are as follows:

- Simple in-band time synchronization
- Low memory overhead
- Low processor overhead
- Small slot size

The purpose of the TDMA protocol for this investigation is to be a key component in behavior primitives as explained in Chapter 6. However, its usage is not limited to this investigation or the automation domain. It is intended to be used in any application where a static communication schedule and small packet size is appropriate. The following sections describe the design and implementation of the TDMA protocol on the Firefly platform.

3.2 Design of the TDMA protocol

The design for the TDMA protocol borrows ideas from a TDMA protocol available on the Firefly, RT-Link [19], which is described in more detail in Chapter 2. This investigation choose to implement a separate protocol instead of use RT-Link for several reasons. First, RT-Link is implemented as task in the Nano-RK OS. This investigation does not use an OS as the complexity of the node level architecture described in Chapter 5 does not warrant it. Additionally, the slot size in RT-Link is 8ms, some of which is required due to using the Nano-RK OS. A smaller slot size can be used if the TDMA is implemented using timer interrupts instead of an OS. Finally, RT-Link is limited in that it must use exactly 32 slots per frame. This is too limiting, as certain applications may require more or fewer slots.

The TDMA protocol design consists of a frame containing one or more slots. Each slot supports a variable sized payload up to a maximum size. The number of slots and the maximum payload size are known by all communicating nodes at

Figure 3.1: TDMA frame

compile time. Each node fixes its own static communication schedule, specifying on which slots it will transmit and receive.

The beginning of each frame is signaled when a dedicated coordinator node transmits a synchronization message. At the beginning of each frame, each node in the network will listen for the synchronization message. The size of the synchronization message is fixed, so when a node receives it the time the message started can be calculated. Given the message start time and the slot width, the start time of the next slot is known. When a node receives the synchronization message is synchronizes its local clock to the time of the next slot. Successive slots in the frame follow every slot duration. At the end of the frame all nodes wait again for the synchronization message. Figure 3.1 shows the layout of a TDMA frame.

Oscillators experience drift over time, causing microprocessors who were at one time synchronized to eventually lose synchronization. To compensate for this in a frame, guard times are added to individual slots. There are two guard times used, one before transmission (TX guard) and one after reception (RX guard). Due

to oscillator drift a receiving node may be ready to receive later than expected. Delaying transmitting nodes compensates for this and allowing late nodes to receive. Similarly, a transmitting node may end a slot late, potentially colliding with the next slot. By forcing silence at the end of a slot this collision is avoided. The RX guard also gives receiving nodes time to process a received packet. The actual durations of the guards are dependent on the microprocessor and oscillator used as well as the duration of a frame.

3.3 Implementation

The TDMA protocol was implemented on the Firefly sensor platform. As an OS was not used, a TDMA task was implemented using a periodic timer interrupt on the microprocessor.

The low-level software to control the CC2420 radio was derived from software available from Nano-RK. Each packet transmitted over the wireless follows the format shown in Figure 3.2. As the maximum packet size the CC2420 radio can support is 128 bytes, the largest payload possible is 117 bytes.

Although the payload size can be as large as 117 bytes, reducing this will result in a smaller slot duration. Considering the application for this TDMA, a small sized payload will suffice. For this investigation, a payload of 15 bytes was chosen on the grounds that the message size of the application is small and messages are infrequent. This gives a total packet size of 26 bytes.

Figure 3.2: Format of a packet. Numbers represent size in bytes, where n is variable length

The CC2420 radio on the Firefly has the option to compute a CRC in hardware of all packet data being transmitted. The CRC used for the computation is $x^{16} + x^{12} + x^5 + 1$, which detects all 1-, 2-, and 3-bit errors [20]. Through experimentation, it was found that although this CRC catches most experienced error cases, there are still a surprising number of errors that go undetected. Given the large time required to compute in software a CRC with better error detecting performance, a software computed checksum was chosen to compliment the radio's CRC. Using the combination of the CRC and checksum, it was found that very few errors pass undetected.

Several factors need to be considered when determining the slot duration. Transmitting a packet over the CC2420 radio requires: 1) an checksum computation and 2) SPI transfer to the CC2420 on the transmitting Firefly, 3) a radio calibration, 4) a wireless transmission including the packet and 7 bytes of overhead from a preamble, start of frame delimiter, and radio computed CRC, 5) an SPI transfer from

the CC2420 on the receiving Firefly along with 6) a checksum verification. Through experimentation, it was found that using the Firefly's 3.68 MHz SPI clock, an SPI transfer of n bytes takes $(13.31 + 4.46n)\mu\text{s}$. A checksum computation, which consists of adding all bytes in a message together, takes $(12 + 1.08n)\mu\text{s}$. For a payload of 15 bytes and packet overhead of 11 bytes, the SPI transfer requires $40\mu\text{s}$, the checksum computation requires $130\mu\text{s}$, a radio calibration requires $192\mu\text{s}$, and a wireless transfer (15 byte payload + 11 byte overhead + 7 byte header and footer) at 250 Kbps requires $1056\mu\text{s}$. Considering both the transmitter and receiver, a complete transmission and reception requires $1588\mu\text{s}$. This value does not include guard times, interrupt overhead, or software overhead.

The appropriate guard times are determined by the relative skew of the Firefly's microprocessors. Through experimentation, it was found that the relative skew between two Firefly nodes is approximately $20\mu\text{s}/\text{second}$, which matches results found in [19]. As this skew is relatively small, after considering software overheads and a small amount of time for application packet processing, the slot duration was decided to be 2ms, where the start guard time is $300\mu\text{s}$ and the end guard time is $282\mu\text{s}$.

Figure 3.3 shows when the various activities and guard times in a slot occur relative to each other, assuming zero timing offset between the sender and receiver. After the last activity is processed, the TDMA task ends, allowing the non-interrupt application to resume execution. The successful packet transmission rates for various frame sizes is shown in Figure 3.4. As the length of the frame increases, the time between synchronizations also increases, causing larger frames to incur more failures

Figure 3.3: Example timing layout of a slot, times not to scale

due to oscillator drift. If a higher success is necessary for a larger number of slots, the guard times can be appropriately increased.

The behavior of the TDMA task is simplistic. Once activated, it will block and wait for the synchronization message from the designated coordinator. When this message is received, the task will wake at the beginning of every slot. If the slot is scheduled for receiving or sending it will perform the appropriate action. Otherwise, the task sleeps until the beginning of the next slot.

Usage of the TDMA task is facilitated through several API interfaces. These interfaces are described below.

3.3.1 Initialization

A node can operate in one of two modes, either as a user or a coordinator. Exactly one node must be a coordinator in the system, who has the responsibility to send the synchronization message at the beginning of the frame. Passing either `TDMA_USER` or `TDMA_COORDINATOR` into the *tdmaInit* interface will initialize the TDMA task.

Figure 3.4: Successful packet transmission rates based on a 26 byte packet in a 2ms slot

After initialization, every slot defaults to an unused state. Each slot to be used must be individually configured using the *tdmaSetSlotMode* interface. This interface accepts a slot number and a mode for the slot, either TDMA_SLOT_RX or TDMA_SLOT_TX. Once the slot schedule is configured the TDMA task is started by invoking *tdmaStart*, which is responsible for starting the TDMA task's hardware timer.

3.3.2 Transmitting

The TDMA task contains no transmission buffers. These buffers must be provided by a user on a per transfer basis. When a transmission on a TDMA_SLOT_TX configured slot is needed, the *tdmaTxPacket* interface can be called, which accepts a pointer to a transfer buffer, the length of the buffer, and the slot to transfer on. At

most one packet can be scheduled for transfer on a given slot, but all slots can have a transfer pending given that each has a unique buffer.

3.3.3 Receiving

Due to limitations in memory, the TDMA task contains a receive buffer for one packet. If a packet is received over the wireless on a TDMA_SLOT_RX configured slot, the packet is copied into the buffer from the radio and stored. Invoking the *tdmaGetReceivedPacket* interface will pass to the user a pointer to the buffer. While the receive buffer contains a packet no further packets will be received. Once the user is finished accessing the received packet, calling *tdmaReleaseReceivedPacket* clears the packet from the buffer, allowing further packets to be received.

CHAPTER IV

BEHAVIOR SPECIFICATION AND ITS EXECUTION

A precise definition of desired behavior is necessary to predictably regulate any system. This problem is particularly challenging in the realm of networked embedded systems because the desired behaviors are based on system-level metrics. Since the nodes execute local tasks and interact with each other in their local neighborhoods, the system-level objectives must be broken down into node-level behaviors. Automating this task is difficult and is a well-known open problem [?]. In this Chapter, the theoretical language UNITY [21] is used as a basis for the specification. While this notation provides the precision necessary at the node-level, it is also amenable to analysis at the system-level, which may be investigated in the future.

We consider node-level specifications at two levels. A higher-level language is used by human users to develop programs for the nodes. To enable efficient execution of this specification, it is translated to a lower-level notation that is used by the expression evaluator runtime discussed in Chapter 5.

4.1 User Level Behavior Specification

The behavior of PLCs is specified in a low-level language called Ladder. Ladder logic is a language which uses a periodic scan architecture described in Chapter 2. The

adaptation of UNITY described in this section was tailored to retain many of the features that are commonly available in the Ladder programming language.

A UNITY program consists of three different sections: *declare*, *initially*, and *assign*. The *declare* section declares the variables used by the program. The *initially* section sets the initial value to each declared variable. The statements to be executed in the program are in the *assign* section. Each statement is an assignment. UNITY contains no flow control, so the statements in a program are evaluated in a random order until no statement when invoked causes a change to the variables. UNITY programs terminate when they effectively converge on a stable state.

The user level specification is similar to UNITY syntax in that variables are declared and initialized in *Variables* and *Initially* sections, respectively, and a *Statements* section contains assignment statements. However, there are four main differences between a UNITY program and a user level specification. First, specifications are deterministic in that statements are always evaluated in the order they are declared. Second, statements consist of a condition and one or more assignments. If a condition, which is a logical statement containing operators, variables, and constants, evaluates to true, then all associated assignments are invoked in the order declared. Third, before evaluating statements an image of the variables is taken and all statements are evaluated sequentially using the same image. Finally, UNITY programs can terminate whereas user level specifications will continue evaluating statements and never terminate.

Figure 4.1: A microcontroller regulates local operations and interacts with its upstream and downstream units via communication channels InCh and OutCh, respectively

As nodes must communicate with other nodes, the user level specification language provides TDMA based communication. Two communication primitives, `commIn` and `commOut`, provide a mailbox abstraction for receiving and sending messages, respectively. An optional *Communication* section declares how a node will communicate over TDMA with other nodes. To correctly communicate, a specification needs to declare the node's ID, which TDMA slots to send and receive on, and which mailboxes are associated with which external nodes. Figure 4.2 shows an example specification for a simple conveyor system shown in Figure 4.1. Following is an explanation of the program.

In statement A2, a unit senses an object at its downstream sensor and sends a message through its downstream communication channel. In A3, if a unit receives a message that indicates an object may arrive, it turns on its Watchdog timer and

```

Program Segment;
Variables
  int          numEntities;
  Timer        mWDT 200; // Timer expires after 200 ms
  dSensor      u;
  dSensor      d;
  dActuator    a;
  commIn       upInCh;
  commOut      upOutCh;
  commIn       downInCh;
  commOut      downOutCh;
end

Initially
  numEntities = 0;
  set mWDT      Stop;
  set u         InActive;
  set d         InActive;
  set a         DeEnergized;
  set upInCh    Empty;
  set upOutCh   Empty;
  set downInCh  Empty;
  set downOutCh Empty;
end

Communication
  LocalID      2; // ID of this node is 2
  upInCh       1; // Messages received from node 1 are read via upInCh
  upOutCh      1; // Messages written to upOutCh are sent to node 1
  downInCh     3; // Messages received from node 3 are read via downInCh
  downOutCh    3; // Messages written to downOutCh are sent to node 3
  Receive      7; // A TDMA slot to receive on (ex: node 1 sends on slot 7)
  Receive      9; // A TDMA slot to receive on (ex: node 3 sends on slot 9)
  Send         8; // TDMA slot to send on (ex: nodes 1 and 3 will receive on slot 8)
end

Statements
  A1: chk a Energized |=
    set a Energized;
  end

  A2: chk d Rising |=
    set downOutCh ReqActOn;
  end

  A3: chk upInCh ReqActOn |=
    set a Energized;
    set mWDT Start;
  end

  A4: (mWDT chk Done) AND (chk u InActive) AND (chk a Energized) AND NOT (numEntities == 0) |=
    set a DeEnergized;
  end

  A5: (chk mWDT Ticking) AND (chk u Rising) |=
    set mWDT Stop;
    numEntities += 1;
  end

  A6: chk u Falling |=
    set upOutCh ReqActOff;
  end

  A7: (chk downInCh ReqActOff) AND (numEntities == 0) |=
    set a DeEnergized;
  end

  A8: chk d Falling |=
    numEntities -= 1;
  end
end

```

Figure 4.2: Example user level behavior specification for a conveyor unit

conveyor belt actuator. In case this message was spurious, or the object did not arrive, the Watchdog timer will expire and the unit will turn off its belt if it is not moving other parts (A4). If an object actually arrives (A5), the unit records the new object. In statement A6, when a unit confirms that the object is fully on its belt, it signals its upstream unit to turn off its belt. When a unit receives a signal to turn off its belt (A7), it checks to ensure that there are no objects on its belt before turning off the belt. In A8, the unit decrements its count when it senses an object leaving its downstream end. Statement A1 serves as a latch, where if any statement energizes the belt, the belt will remain energized until a statement de-energizes it.

4.1.1 Important Features

The user level specification language has several important features which distinguish it from other languages used for embedded systems, such as C.

To better facilitate working with discrete devices, several special data types are available. These data types, dSensor and dActuator, abstract an external discrete sensor or actuator, respectively. An actuator can be asserted as two states, either energized or de-energized. A sensor can be in either of four different states, either inactive, active, rising (transition from inactive to active), and falling (transition from active to inactive).

Many programming languages that deal with asynchronous inputs need special consideration from users to avoid race conditions. The statements in the user specification language are evaluated in order using identical, locked inputs. This

guarantees that the statements are evaluated atomically, preventing race conditions which arise from evaluating logic while inputs are changing.

Another feature not common in typical embedded languages is an implicit concept of time. The user specification language allows timers to be declared and their timeouts specified. The advancement and firing of the declared timers is handled automatically, removing the need to manage timers manually in software.

A final feature available in higher level languages such as Erlang and Scala is the use of a mailbox abstraction to facilitate communication. The user specification language assumes the use of TDMA based communications between nodes. Once a node's specification declares the node's ID, which slots to use, and associates mailboxes with sending or receiving messages, all wireless communication is handled automatically.

4.2 Device Level Behavior Specification

The user level specification is organized into sections and each rule is expressed in infix notation, where an operator appears between its operands. Expressing a specification in this form is simple for a human to interpret. However, both the notation used for the rules and the layout of the specification can be modified such that a device can more efficiently interpret and evaluate the specification. The specification used directly by the devices is simplified, as described below.

It is advantageous to evaluate a statement using as little time as possible. To facilitate quick evaluation, each statement is converted to postfix notation, where

operands occur before operators. The advantage to this notation over infix notation are two fold. First, the evaluation of a statement can be accomplished using only a single stack. Second, postfix notation contains no braces which reduce size and complexity.

The layout of the user specification consists of sections, where each section starts with a title and ends with the keyword *end*. Reading a section consists of reading expressions until the terminating *end* is found. As long as the order of the sections is maintained, the complexity of reading a section can be reduced by eliminating the title and terminating *end*, and marking each expression with a keyword identifying its section of origin. Additionally, the variable declarations and initializations can be condensed into one expression.

Memory usage is a concern, as it ultimately limits the size of applications. Careful consideration needs to be made to ensure memory is being used efficiently. The user level specification does not set a maximum length for variable names. Usually variables are labeled to improve readability. However, a variable name is only useful to a device as an identifier. Therefore, to reduce memory usage on a device each variable name is reduced to a simple identifier. Each variable name is converted to a string starting with a letter and ending with a number. The letter signifies what type of variable the name represents, for example 's' refers to a dSensor and 'i' refers to an integer. The number uniquely identifies the variable from others of the same type. This number starts at 0 and increments for each variable of the same

```

var int i00 = 0
var bool b00 = false
var Timer t00 set Stop 200
var dSensor s00 set InActive
var dSensor s01 set InActive
var dActuator a00 set DeEnergized
var commIn c00 set Empty
var commIn c01 set Empty
var commOut o00 set Empty
var commOut o01 set Empty
comm LocalID 2
comm recv 7
comm recv 9
comm send 8
comm neighbour c00 1
comm neighbour c01 3
comm neighbour o00 1
comm neighbour o01 3
A1: a00 Energized chk |= a00 set Energized ;
A2: s01 Rising chk |= o01 set ReqActOn ;
A3: c00 ReqActOn chk |= a00 set Energized ; t00 set Start ;
A4: t00 Done chk s00 InActive chk AND a00 Energized chk AND i00 0 == ↵
    NOT AND |= a00 set DeEnergized
A5: t00 Ticking chk s00 Rising chk AND |= t00 set Stop ; i00 += 1 ;
A6: s00 Falling chk |= o00 set ReqActOff ;
A7: c01 ReqActOff chk i00 0 == AND |= a00 set DeEnergized ;
A8: s01 Falling chk |= i00 -= 1 ;

```

Figure 4.3: Example device level behavior specification for a conveyor unit; the ↵ symbol means the line continues unbroken

type. When a variable name is compressed, the change is echoed in both variable initialization/declaration expressions and statements.

To apply these modifications, the device level specification contains three different type of expressions: variable declaration/initialization, communication configuration, and statements. Each variable expression is identified by a preceding *var* keyword, and contains both a declaration and initialization. Communication expressions are identified by a preceding *comm* keyword. Statements are identified by a name followed by a colon, where each name must not be a keyword. Each Statement contains a condition in postfix notation and one or more actions in infix notation, each separated by a semicolon. Figure 4.3 shows the device level specification for the user level specification in Figure 4.2.

Figure 4.4: Extending the scan architecture of PLCs

4.3 Device Specification Evaluator

The device specification evaluator is separated into two logical tasks, one to evaluate expressions and one to manage wireless communications. The communication task is synchronized to a TDMA schedule and occurs in regular intervals, where the evaluation task executes continuously. Passing messages to and from the tasks is facilitated by common message exchange buffers.

Much like most industrial controllers, the evaluation task achieves predictable operation using a periodic scan architecture. Sensed values are copied into an input image table to ensure that the values do not change during the computation. Temporary values are held in the output image table until the end of the evaluation and presented to local actuators. These actions repeat periodically. Before each evalua-

tion of the logic, values of the output image table are restored to their default state. The operation of the expression evaluator is shown in Figure 4.4.

The blocks labeled IPB and OPB are temporary buffers to hold input and output values, respectively, to facilitate interaction with the physical world. In addition to values from local devices, nodes rely on a communication channel to receive or send sensed and computed values from other nodes. The type identifier records whether a particular value is a local sensor, local actuator, a value received, or a value to be sent over a communication channel.

The device level specification is stored in each node as a table with each row corresponding to a statement consisting of a condition and set of actions. The evaluation task will compute the value of a condition, either True or False, based on the values in the input image table. If a condition evaluates to True, each associated action is executed and the results stored in the output image table.

CHAPTER V

NOTE LEVEL ARCHITECTURE

This chapter presents a design for a specification execution engine that was implemented for this investigation. This design was implemented on the FireFly platform available from Carnegie Mellon University. The FireFly uses an ATmega1281 microprocessor - an 8-bit processor with 8K Bytes of RAM operating at 7.37 MHz - with both an integrated CC2420 wireless transceiver and a multimedia card (mini-SD) reader that interface with the processor over an SPI bus.

5.1 Specification Program Input

A device level specification is simply a string of text which is read by the expression evaluator, interpreted, then executed. A specification is loaded onto a mini-SD card, then inserted into the FireFly's multimedia card reader before startup. When activated, the FireFly accesses the mini-SD card, reading the entire specification and storing a representation into RAM before execution. If the power on the FireFly is cycled, the specification is read again from the mini-SD card.

5.2 Specification Memory Storage

A device level specification requires storing two types of tables in memory: one for runtime variables and one for statements and associated actions. As memory on the FireFly is limited, all table allocation is done statically at compile time using several configurations specifying the maximum number of supported rules, tokens, etc. In doing so, it can be guaranteed at compile time that a FireFly can store the entire specification and execute properly.

5.2.1 Variable Table

The specification language provides for several different variable types. These types include discrete sensors, discrete actuators, integers, communication, and timers. Each variable uses the same representation in memory, which is shown in Figure 5.1.

At runtime variables are categorized into three categories depending on how they are used:

- Input variables: sensors, incoming communications, timers.
- Output variables: actuators, outgoing communications.
- Standard variables: integers, booleans

Each variable category is maintained in its own list structure. A second copy of each list is present when the variables are locked during expression evaluation.

```

typedef struct
{
    char name[MAX_VARNAME_LENGTH]; // Name of the variable.
    VariableType type; // The type of variable represented.
    union // The value of the variable.
    {
        int intValue;
        bool boolValue;
        TimerValue timerValue;
        dSensorValue dSensorValue;
        dActuatorValue dActuatorValue;
        commInValue commInValue;
        commOutValue commOutValue;
    } value;
    TimerContainer * timerContainer; // This is filled in only if the variable is a timer.
                                     // This representation is necessary due to the large
                                     // space that timers require.
} Variable;

// Note that a static number of timers are allocated at compile time. During parsing
// timer variables are given a timer struct.

typedef struct
{
    uint32_t startTimeMs; // System time that this timer started counting.
    uint32_t maxTimerValueMs; // Maximum value this timer will count to
} TimerContainer;

```

Figure 5.1: Definition of a variable

5.2.2 Expression Table

Each specification consists of one or more statements. Each statement consists of a rule followed by one or more actions. Rules and actions are decomposed into a sequence of tokens. A token is the smallest unit of an statement and can represent a symbol (e.g. ==, NOT, AND), a constant, or a variable. Figure 5.2 shows the definition of a token. As a token is used to represent three different entities, a union is used to reduce memory.

Each statement is stored in a list structure in the order declared in the specification.

```

typedef struct
{
    TokenType type; // Single byte labeling what the union represents.

    // The union contains data for either a symbol, constant, or variable.
    union
    {
        char variableName[MAX_VARNAME_LENGTH]; // Represents a variable.
        SymbolType symbol; // Represents a symbol
        struct // Represents a constant
        {
            VariableType type; // Labels what type of constant this is.
            union
            {
                int intVal;
                bool boolVal;
                TimerValue timerVal;
                dSensorValue dSensorVal;
                dActuatorValue dActuatorVal;
                commInValue commInVal;
                commOutValue commOutVal;
            } value;
        } constant;
    };
} Token;

```

Figure 5.2: Definition of a token

5.3 Mapping Execution to the Environment

The FireFly and expression evaluator have no concept of its environment or what the overall goal of a specification is. There is no implicit mapping between sensor and actuator variables and the environment. To eliminate this problem, there is an explicit mapping between each sensor and actuator variable and a pin on the FireFly. There are two pin headers on the Firefly, one labeled *UARTS* and one labeled *ADC*. The *UARTS* header is used to represent actuators and each pin is given an index, likewise for the *ADC* header for sensors. Each actuator and sensor variable is represented by the letter ‘a’ or ‘s’ respectively and an index. This index on the variable names maps directly to the index on the individual pins. As the variable indicies are assigned

based on the order they are declared in the specification, the first sensor variable maps to the first sensor pin, and so on.

5.4 Specification Execution

Once the entire specification is read and stored into memory, the mote will begin executing the specification. The execution is facilitated by using two tasks, one for expression evaluation and one for communication. No operating system is used on the mote, so the communication task is implemented using timer interrupts. To reduce overall complexity, the communication task also maintains system time to help manage Timers.

5.4.1 Evaluation task

The evaluation task implements the node-level execution architecture shown in Figure 4.4. All italicized terms in this paragraph correspond to blocks in the figure. This task executes *CommsIn*, *LocalSense*, *Evaluate*, *LocalActuate*, and *CommsOut* in this order. the actual number of actions in the engine determines the period of the task - the next scan starts as soon as the preceding scan completes. *CommsOut* posts packets to the communication task, and *CommsIn* receives packets from the communication task, both via shared buffers in memory.

5.4.2 Communication task

Reliable communication channels between nodes is a critical component in networked embedded automation systems. A TDMA-based scheme described in Chapter 3, is

used. Each frame comprises 20 communication slots and 1 synchronization slot. Each slot has a duration of 2ms, for a frame duration of 42ms. Node synchronization is accomplished by a dedicated node sending a beacon in the synchronization slot, where all nodes who receive the beacon use its timing information to become synchronized for the remaining of the frame. The communication task is implemented using periodic hardware timer interrupts. A 20 slot frame allows for a packet success rate of 99.8%.

CHAPTER VI

SYSTEM LEVEL CONSIDERATIONS

The spatial and temporal separation of functionality in automation systems allows designers to modularize the design, operation, and maintenance of these systems. Building on this idea, networked embedded automation systems can be composed using modular units. However, it is necessary to understand the limits of performance that can be achieved by individual units and also in the context of coordinated operations of two or more units that deliver system objectives. This chapter presents several local and system level behavior primitives necessary for local and coordinated actions across multiple units.

Nodes in a networked embedded automation system typically energize local actuators. For this investigation, only discrete sensors and actuators, i.e., sensors that provide digital values and actuators that respond to digital commands, are considered¹. Each node executes the logic necessary to compute values for its actuators based on locally sensed or computed values, similar values obtained from other nodes in the system, or a combination of such values. Towards this end, several primitives are considered for networked embedded automation. These primitives are described in the following sections.

¹When non-discrete devices are considered, issues of control including non-zero time required to change state, holding time, and cycle time must be accounted for.

Figure 6.1: Periodic scan time primitive example

6.1 Periodic Scan Time

The periodic scan time is the simplest primitive, and represents the duration of a single periodic scan. The number of rules executed and the presence of background communication can change this duration. This primitive provides a lower bound time for any response the system can compute. A simple example of this is using a timer to trigger a future event, as seen in the timeline in Figure 6.1. If the system starts a timer at time 0 with a timeout duration of t' , the response to the timeout will occur between times t' and $t' + 2\Delta$, where Δ is the loop execution time. The reason the timeout may not be responded to at time t' is the system may be executing a scan started earlier and is unaware of the timeout until the scan starts again.

6.2 Local Response

A local response is the duration between a change in local (sensed or computed) values and energize a local discrete actuator using logic evaluated locally. Figure 6.2 gives an example of this primitive, where a node is connected to a part detection sensor and a conveyor actuator. At time t_0 the actuator is energized, pushing a part toward the sensor. When the part encounters the sensor the node will respond by deenergizing

Figure 6.2: Local response primitive example

the conveyor. At time t_1 the response completes. As there is a finite delay in the response, the part continues to move along the conveyor until it is deenergized.

6.3 Remote Response

The remote response represents the energizing of a local discrete actuator using local logic in response to a non-local (sensed or computed) value received via a single hop communication. This represents a “push” model for data, where a message is sent to neighboring nodes when the local state changes. Figure 6.3 shows an example of

Figure 6.3: Remote response primitive example

this where a node n_1 is connected to a switch as a discrete sensor, and a node n_2 is connected to a light bulb as a discrete actuator. At time t_0 node n_1 detects that the sensor changes state and sends a wireless message to node n_2 at time t_1 . At time t_2 node n_2 responds to the wireless message by energizing its actuator.

6.4 Request/Response

The “pull” model for data is represented in the request/response primitive, where nodes request information from neighboring nodes. This primitive represents a node requesting non-local values from a neighbor a single hop away, and energizing a local actuator using local logic based on the neighbor’s response. An example of this is in

Figure 6.4: Request/Response primitive example

Figure 6.4. Node n_1 is connected to an actuator which is deenergized. At t_0 node n_1 sends a wireless request to node n_2 . Node n_2 responds at time t_1 , and based on the data in the response node n_1 decides to energizes its actuator, which occurs at time t_2 .

6.5 Publish/Subscribe

When the remote response primitive involves more than one receiving node, it is important to characterize how close in time the responses of the receiving nodes will be computed. The publish/subscribe primitive captures this. This primitive describes the energizing of a local discrete actuator in one or more subscriber nodes, using local

logic, based on non-local values that are received over a single hop communication channel from a single publisher node. Figure 6.5 shows an example of this where n_1 is a publisher and nodes n_2 and n_3 are subscribers. At time t_0 the subscriber node detects a change in its local sensor and broadcasts a message to all subscribers at time t_1 . The subscribers will receive the message at the same time, but may not response to it at the same time due to where they are in their current scan cycle. At time t_2 and t_3 the first and second subscriber responds to the message, respectively.

6.6 Multihop Publish/Subscribe

Some activities may require that a message be relayed several hops to reach its destination. The multihop publish/subscribe primitive represents the energizing of a local discrete actuator in one or more subscriber nodes, using local logic, based on non-local values that are received via multihop communication from a single publisher node. Figure 6.6 shows an example where node n_1 is a publisher, node n_k is a subscriber, and a message must be relayed by each node $n_i, 2 \leq i \leq k - 1$. At time t_0 node n_1 senses a change in its local sensor and sends a message toward its subscriber at t_1 . At each time t_i , node n_i relays the message one hop closer to the subscriber. At time t_{k-1} the subscriber node receives the message and its local actuator is energized at time t_k .

Figure 6.5: Publish/Subscribe primitive example

Figure 6.6: Multihop publish/subscribe primitive example

6.7 Multihop Request/Response

The multihop request/response primitive is the multihop version of the request/response primitive, where a node will request non-local values from another node via multihop communication, and will energize its local discrete actuator using local logic based on the response. Figure 6.7 demonstrates the primitive where node n_1 sends a request to a node n_k , k hops away. At time t_1 node sends the request toward the requesting node and node n_2 receives the request. The request is then forwarded, hop by hop, until it arrives at node n_k at time t_{k-1} . A response is computed and transmitted at time t_k , which is forwarded until it arrives at node n_1 at time t_{2k} . A response to the message is computed which energizes the local actuator at time t_{2k+1} .

Figure 6.7: Multihop request/response primitive example

CHAPTER VII

IMPLEMENTATION RESULTS

This chapter presents experimental results of the implemented system on the FireFly with respect to the six primitives. Following, results from deploying a set of FireFly motes to regulate the behavior of a model microfactory are shown.

7.1 Performance Evaluation

A performance profile of the seven primitives helps to quantify the latency and jitter in the system. First, the evaluation engine was profiled, without any sensor inputs, actuations, or communication, as seen in Figure 7.1. The number of statements was varied between 10 and 60. For each set of statements, the evaluation engine was executed for 10,000 iterations. The granularity of the time measured was 1 μ s. The jitter is stable as the number of rules increase, as the only background activity is a software clock. In the following experiments the computational load on the nodes is progressively increased by bringing in sensing, actuating, and communication activities.

To understand the effects that communication has on performance, again the evaluation engine was run with a varying number of statements from 10 to 60 as before, but with background communication where transmission and reception is

Figure 7.1: Execution time of evaluation engine with no sensor inputs, actuations, or communication

schedule for one slot each. The resulting execution time is shown in Figure 7.2. As communication interrupts the execution of the evaluation engine, adding communication results in longer execution times and increased jitter. Note that even executing 60 rules the jitter is less than 1 ms.

The latency and jitter shown in Figures 7.1 and 7.2 are based on a hardware timer in the node¹. For the remaining experiments in this section, latency measurements were obtained using an oscilloscope. Twenty samples were collected for each measurement. Sensor events were injected from a signal generator and the precise latency for each experiment is specified using the notation in Table 7.1. All commu-

¹The latency reported is the average over all samples and jitter is the standard deviation. In the figures, Avg+stdev and Avg-stdev are shown to provide an indication of the deviation from average values.

Figure 7.2: Execution time of evaluation engine with no sensor inputs or actuations, with background communication

Table 7.1: Notation

Symbol	Meaning
$u \uparrow$	Sensor u is Active
$u \downarrow$	Sensor u is Inactive
$u \nearrow$	Sensor u transitioned from Inactive to Active
$u \searrow$	Sensor u transitioned from Active to Inactive
$\tilde{a}$	Actuator a is Energized
$\dot{a}$	Actuator a is De-Energized

Figure 7.3: Response time of evaluation engine with sensor input and actuation, but no communication

communications use a 21 slot frame, where each slot is 2ms, and each node transmits on only one slot per frame.

Figure 7.3 shows the performance of the Local Response primitive. The node energized an actuator a whenever a sensor s detected an event. The latency, $t(\tilde{a}) - t(s \uparrow)$, is shown in the figure. Depending on the number of statements, the evaluation engine requires different durations of time to complete its evaluation following the scan cycle shown in Figure 4.4. The jitter in Figure 7.3 is due to when the sensor event occurs in the scan cycle. Thus, in the worst case, the current scan cycle must complete before a sensor event can be used to energize an actuator in the next scan cycle.

Figure 7.4: Response time of evaluation engine with sensor inputs and background communication

Figure 7.4 shows the change in the performance of the Local Response primitive when the node was communicating with another node in the background. In this experiment, there are two tasks executing - the communication task is running periodically. Thus, the TDMA task is the primary source of the increase in jitter. As the number of statements increase, the time required in the evaluation engine increases, and hence the jitter increases. Nevertheless, note that the jitter is under 8ms even when the node is executing 60 actions.

Figure 7.5 shows the performance of the Remote Response primitive. Two nodes, N_1 and N_2 , were used with an actuator a on N_1 . Upon starting, N_2 waited for a local button b to be depressed; a message was sent to N_1 with a predetermined

Figure 7.5: Response time of 1-hop 1-way communication from a sensor to an actuator over a wireless link

value. N_1 energized a upon receiving the message. The latency, $t(\tilde{a}) - t(b \uparrow)$, is shown in the figure. Notice, that the jitter is around 16ms and is relatively stable. N_1 is only scheduled to send one message in a 42ms frame. Thus, the jitter is likely caused by when b is actually depressed relative to the TDMA schedule.

Figure 7.6 shows the performance of the Request/Response primitive. Two nodes are used, N_1 and N_2 , with an actuator a and button b on N_1 . Upon starting, N_1 waited for b to be depressed; a message was sent to N_2 , and a response from N_2 was received at N_1 . N_1 energized a upon receiving the response. The send slots for N_1 and N_2 are separated by 20ms. The latency, $t(\tilde{a}) - t(b \uparrow)$, is shown in the figure. Notice that there is more jitter compared to one-way communication. The reason is

Figure 7.6: Response time of a request/response primitive with 2-way communication over a 1-hop network

the execution of the evaluation engine may or may not have completed before the scheduled TDMA slot for sending a response. This can be seen at 30 rules, where the time to compute a response is also the time between slots, 20ms. As the figure shows, sometimes N_2 computes the response in time, and sometimes it does not, resulting in waiting an entire frame to try again. Similar behavior also occurs at 60 rules.

Two aspects are important for the Publish/Subscribe primitive. The performance of the Remote Response primitive implicitly affects the Publish/Subscribe primitive. In addition, it is important to understand the effects of propagation. An experiment was designed with three nodes, N_1 , N_2 , and N_3 . N_2 had actuator a_2 and N_3 had actuator a_3 . Figure 7.7 shows the latency, $|t(\tilde{a}_2) - t(\tilde{a}_3)|$ when N_1 ini-

Figure 7.7: Actuation response times between two nodes under variable load

tiated the actuations with a message. The two numbers along the x-axis represent the number of actions in N_2 and N_3 , respectively. Such a situation is useful when a node is attempting to coordinate actions on multiple nodes, for example in multi-axis synchronization.

Next, the scalability of the Publish/Subscribe and Request/Response primitives was considered. Figure 7.8 shows the performance of the Multihop Publish/Subscribe primitive. Nodes $N_1, N_2, \dots, N_k$, were used where $k \in \{4, 5, 6, 7, 8\}$, and node N_i contains $8+2i$ actions. A button b was on N_1 , and actuator a_k was on N_k . Send slots are separated by 14ms. The figures present the latency $t(\tilde{a}_k) - t(b \uparrow)$. While the linear rise with the increase in hops is as expected, the jitter is high at around 40ms. The reason for this is similar to that for the Request/Response prim-

Figure 7.8: Multihop Publish/Subscribe scalability

itive, where nodes may or may not compute a response before their designated slot. The scalability of the Multihop Request/Response primitive, shown in Figure 7.9, is similar. In both figures, the spread between the minimum and maximum response time grows slightly (from 100ms to 150ms), due to adding nodes with larger number of actions. On average 2-4 nodes miss their slot and delay an extra frame per trial.

7.2 Model Factory Deployment

To demonstrate the evaluation engine coordinating the behavior of a complete system, Firefly motes with appropriate rule sets were deployed to regulate a model microfactory. This microfactory, available at the Complex Engineered Systems Laboratory at the University of Akron, physically models a discrete sensor/actuator assembly

Figure 7.9: Multihop Request/Response scalability

line. A drawing of the microfactory setup is shown in Figure 7.10. The microfactory mimics the creation of painted wooden parts. Grey tracks show conveyor belts, and green triangles represent discrete part sensors. Starting at the top left of the figure and moving clockwise, the modules drawn are a raw part input bin, workcell for vertical and horizontal milling, a segment, a crossover for diverting parts onto different paths, a paint station, a turnaround, a washer installer, and a completed part output station. Each of these modules is regulated by one or more Firefly nodes. A part will enter the system via the raw part input bin, and will move from module to module until it reaches the output bin. Part transfers from module to module are facilitated via wireless handshake messages.

Figure 7.10: Drawing of the microfactory

The primitives mentioned earlier map to the behaviors of the microfactory. Following is an example of how each primitive manifests itself.

Evaluation Time: Gives lower bound for timer resolution. Example is moving a block to another module, using a giveup timeout.

Local Response: Stopping a part when it is in position at a sensor.

Remote Response: Receiving a message from a downstream neighbor and de-energizing the local conveyor.

Publish/Subscribe: Work-cell instructing the vertical and horizontal mills to process their parts.

Request/Response: Negotiation for transferring parts from one unit to another.

Multihop Request/Response: Crossover requests via multiple hops if the turnaround is currently enabled. If not, parts are directed only to the output bin.

Multihop Publish/Subscribe: Output bin instructs input bin not to add parts to the system when the output bin is disabled.

CHAPTER VIII

CONCLUSION

Automation systems today have several shortcomings, which include the ability to detect only well-known faults, the inability to leverage current distributed system advances, and communications limited to mode management, transfer of control, and status reporting. To overcome these problems, one potential solution is to augment these automation systems with mote-class devices with low-power wireless transceivers by integrating them into the sensor and actuator systems. Doing so will allow decentralized control that can be more adaptive and responsive to the changing needs of applications. Such augmentation and integration presents several challenges, such as showing that the resulting system is robust, predictable, reliable, and safe. The body of knowledge necessary to address these challenges is referred to as Networked Embedded Automation.

This thesis addressed the challenge of predictability in Networked Embedded Automation. As predictability is a contextual attribute, it was considered at three levels: system architecture, local behaviors, and interactions between nodes. The system architecture was a combination of two elements, a TDMA communication backbone to support inter-node wireless communication and a expression evaluator similar in behavior to a Programmable Automation Controller's scan cycle. Both

elements are predictable as they are periodic and the number of activities performed in each period are bounded. The behavior of each node is described using a specification language similar to the UNITY language. This architecture was implemented on the Firefly platform. When powered on, each Firefly will read a simplified version of its specification from an on-board SD card, load the specification into memory, and repeatedly execute the specification.

To demonstrate predictability, a set of behavioral primitives were defined which represents various simple behaviors that can manifest in automation systems, from local behaviors to behaviors across several nodes. Evaluating these primitives found that the Firefly platform can reliably regulate actuators when the application can tolerate a jitter of 10ms. For applications that can tolerate a jitter of 16ms and a latency of 170ms, singlehop wireless links can be applied. Multihop wireless links of up to 8 hops can be used when applications can tolerate a jitter and latency of 45ms and 600ms, respectively. Applications such as material handling conveyors and critical infrastructure surveillance are good candidates for such systems.

The resulting system on the Firefly platform was deployed onto a model microfactory to regulate its behavior. This microfactory is a physical simulation of a discrete sensor/actuator assembly line. The defined behavioral primitives map into the operations performed by the assembly line, demonstrating how the primitives can be used in practical systems.

BIBLIOGRAPHY

- [1] N. Hayslip, S. Sastry, and J. Gerhardt. Networked embedded automation. *Assembly Automation*, 26(3):235–241, 2006.
- [2] Control Magazine. 50 top automation companies, December 2009.
http://www2.emersonprocess.com/siteadmincenter/PM%20Articles/Control12_09_Top50.pdf.
- [3] Rockwell automation website. Accessed March 1, 2010.
<http://www.ab.com/programmablecontrol>.
- [4] J. Hill, M. Horton, R. Kling, and L. Krishnamurthy. The platforms enabling wireless sensor networks. *Communications of the ACM*, 47(6):41–46, June 2004.
- [5] A. Eswaran, A. Rowe, and R. Rajkumar. Nano-rk: An energy-aware resource-centric operating system for sensor networks. In *IEEE Real-Time Systems Symposium*, December 2005.
- [6] Crossbow mpr-mib users manual. Revision A. http://www.xbow.com/Support/Support_pdf_files/MPR-MIB_Series_Users_Manual.pdf.
- [7] Telos datasheet. Revision 1. www.ece.osu.edu/~bibyk/ee582/telosMote.pdf.
- [8] Tinynode datasheet. Revision 1.1. http://www.tinynode.com/uploads/media/SH-TN184-103_rev1.1.pdf.
- [9] Firefly datasheet. Revision August 20, 2007.
http://www.nanork.org/raw-attachment/wiki/Documentation/FireFly_datasheet.pdf.
- [10] Cc1000 datasheet. Revision SWRS048A. focus.ti.com/lit/ds/symlink/cc1000.pdf.
- [11] Cc2420 datasheet. Revision SWRS041B. focus.ti.com/lit/ds/symlink/cc2420.pdf.

- [12] Sx1211 datasheet. Revision 7. <http://www.semtech.com/images/datasheet/sx1211.pdf>.
- [13] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient mac protocol for wireless sensor networks. In *Proceedings of the IEEE Infocom*, pages 1567–1576, New York, NY, USA, June 2002. USC/Information Sciences Institute, IEEE.
- [14] T. van Dam and K. Langendoen. An adaptive energy-efficient mac protocol for wireless sensor networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 171–180. ACM, 2003.
- [15] A. Warrier I. Rhee, M. Aia, and J. Min. Z-mac: a hybrid mac for wireless sensor networks. In *SenSys*, pages 90–101. ACM Press, November 2005.
- [16] J. Degesys, I. Rose, A. Patel, and R. Nagpal. Desync: self-organizing desynchronization and tdma on wireless sensor networks. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 11–20. ACM, 2007.
- [17] V. Rajendran, K. Obraczka, and J. Garcia-Luna-Aceves. Energy-efficient collision-free medium access control for wireless sensor networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 181–192. ACM, 2003.
- [18] L. van Hoesel and P. Havinga. A lightweight medium access protocol (lmac) for wireless sensor networks: Reducing preamble transmissions and transceiver state switches. In *First international conference on networked sensing systems*, 2004.
- [19] R. Mangharam, A. Rowe, and R. Rajkumar. Voice over sensor networks. In *27th IEEE Real-Time Systems Symposium (RTSS)*, December 2006.
- [20] P. Koopman and T. Chakravarty. Cyclic redundancy code (crc) polynomial selection for embedded networks. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*, page 145, Washington, DC, USA, 2004. IEEE Computer Society.
- [21] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison Wesley, 1988.