Logic-based programming for wireless sensor-actuator networks

Yizhi Wu, Anthony Rowe
Electrical and Computer Engineering Department
Carnegie Mellon University
Pittsburgh, PA 15217
Email: {yizhi,agr}@ece.cmu.edu

***Abstract*—In this paper we present SAN-Logic, a lightweight logic-based programming paradigm that enables the dynamic progammability and configuration of sensor-actuator interactions in wireless sensor networks used to support Cyber-Physical Systems (CPS). Our goal is to simplify complex CPS design by providing a structured model of interactions that can be automatically mapped and deployed to a sensor-actuator network in an efficient and scalable manner. In contrast to sensor networking paradigms that distribute an application into individual sub-programs, SAN-Logic models the system as a set of boolean expressions which can be partitioned across the network like gates in a circuit. The user defines interactions as timed asynchronous sequential logic expressions [1] with sensors and actuators representing the inputs and outputs of the system.**

This approach is highly scalable since once deployed each interaction takes place as a sequence of independent and asynchronous events. This allows SAN-Logic to operate in a fully distributed manner without a central authority. Using this framework, optimization takes place across multiple tasks enabling sharing of resources within the network which will be an important part of future CPS. Redundant routes and the stateless nature of combinational logic (along with periodic state update messages) allow the system to easily cope with packet-loss and failed nodes. A major benefit of this approach is the ability to leverage existing hardware design and synthesis tools used by the VLSI design community. We demonstrate how boolean manipulation of the logic can be used to alter the mapping of expressions onto the network and hence can be used for optimization and verification. We provide an approach using logic simplification and mapping that reduces message passing by factoring common terms across different data paths within tasks and placing intermediate terms such that they benefit from shorter paths. In complex systems, we see on average a 40% reduction in message passing as compared to an implementation that does not optimize communication patterns within and across tasks.

I. Introduction

Wireless sensor networks provide a means for an easy-to-deploy sensing and communication infrastructure making them ideal solutions for Cyber-Physical Systems (CPS) like those found in building and home automation. Nodes equipped with sensors and actuators can be configured to perform tasks ranging from lighting control to HVAC regulation. Modern building HVAC systems are comprised of hundreds of sensor and actuator elements that communicate over expensive wired buses. Often times the interactions between these elements must be customized based on unique building configuration parameters or user requirements. In order for wireless sensor networks to become a practical tool for CPS environments, a framework is required for designing and deploying networks that provides the ability to link sensors and actuators together in a highly customizable manner. A user should be able to describe the desired functionality of the system in a high-level and intuitive abstraction that automatically gets mapped to the low-level operations implemented by each sensor node.

The current approach for programming sensor networks typically involves adding individual tasks that are tied together through explicit message passing. Often times the application code ends up being tightly integrated with the majority of the functionality focused on the networking layer. This works well for networks designed to perform data collection tasks like sampling environmental sensor values but makes it quite complex and time-consuming to implement CPS applications with unique in-network interactions. Even simple local interactions require significant effort to manage and deploy given scale, failure-prone links and heavy resource constraints. Macro-programming refers to the notion that a group of devices can be programmed as a collective rather than manually creating a unique software image for each node. In order to support a high-level description of sensor and actuator interactions that can be decomposed and distributed across a CPS network, it is important to have a simple yet powerful model that can be used to merge, validate and optimize large sets of interactions.

In this paper, we present SAN-Logic (Sensor-Actuator Network Logic), a design paradigm based on logical programming used to build CPS applications. We adopt a dataflow-centric architecture where changing values in the system force the recalculation of other variables. Interactions are represented as boolean expressions with the approach enabling the system to be managed and optimized using existing VLSI and control automation tools. This programming model is very similar to the most common HVAC and automation programming languages used to configure Programmable Logic Controller (PLC) and SCADA systems including Ladder Logic Diagrams, Function Block Diagrams, Structured Text and Sequential Function Charts. In fact, we show an example of how Ladder Logic can be directly translated into SAN-logic. Using a sensor network to operate as the communication medium for an automation system has challenges associated with packet loss, the increased potential for node failure and the extremely resource constrained nature of the devices with respect to

energy and computation power. To address these problems, we use techniques like adaptive periodic state updates for nodes with poor links and multi-path routing with term combining to reduce message traffic. There are many situations that will always require wired solutions due to stringent reliability and timing requirements, however sensor networks are a cost effective solution for augmenting existing CPS with extra sensing and non-critical control components.

SAN-Logic is extremely lightweight and can operate on top of an existing sensor networking infrastructure. For example, if a system already has a means of routing messages and performing global data collection, SAN-Logic can be run on top of this network as a service to enable customizable sensor-actuator interactions. Each interaction is an independent sequence of asynchronous events that once configured can operate without a central authority. This makes the solution highly scalable across large sensor networks since different subsections can run independently from their surrounding nodes. Optimization of these networks utilizes existing VLSI tools that are designed to optimize circuits with tens of millions of gates. In order to maintain the simplicity and small footprint of the engine running on each node, SAN-logic operates on boolean expressions. It is possible to use multiple signals to create a "bus" in order to send non-binary values, however this makes signal sharing and wide-scale optimization more difficult.

The interactions between sensors and actuators in SAN-Logic are specified independent of the underlying physical topology. An initial set of optimizations and validations can be performed on the logical description combining together multiple sub-tasks. Figure 1 shows a typical example of various sensors and actuators that might compose a larger system. In this case, we see part of an HVAC system where there might be many sensor and actuator interactions that are functionally independent. The combination of a temperature and motion sensor in one room might need to trigger a nearby air blower. Similar interactions might be required in neighboring rooms. In some cases, sub-expressions could be shared among independently configured interactions. The same temperature and motion sensors might be used in conjunction with a door position relay to disable lights. Figure 2 shows a simple circuit representation of interactions found in Figure 1 where the letters correspond to sensor and actuator data. Here we see two expressions, one that governs an air conditioning unit and one that controls an alarm. It is easy to see that the $a + b$ input terms can be factored out and reused between tasks. Before deployment, a synthesis step is used to map the interaction logic onto a given (externally captured) network topology. At this point, topology-specific optimizations on the placement of logic components. In the previous HVAC example, the physical placement of common terms will have a significant impact on where messages must be routed. By combining and placing these sub-expressions in advantageous locations, we can optimize various QoS properties such as latency and energy consumption. Finally, individual portions of logic are dispatched to each sensor node where a logic

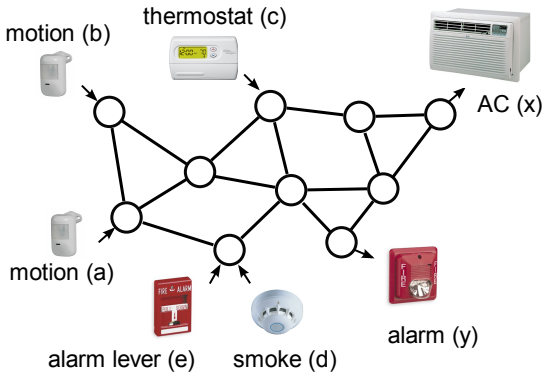


Fig. 1. Example scenario of various sensors and actuators that could be combined to support different applications. Motion detectors can be used to identify occupied and unoccupied regions of a building. An air conditioning system can adjust its performance based on occupancy along with temperature sensors. A fire system could also integrate with temperature and motion sensors to include security detection features during after hour operations.

engine monitors local state, waits for incoming signals and performs any local computations.

The physical component of CPS makes it important to capture not only the data flow of the system but also timing properties. We introduce the notion of *signals* that capture the value and timing parameters of variables. Signals also define the policies and timing values used when data need to propagate. Temporal interactions in the system can be specified using relative or absolute timing attributes on signals. Relative delay-lines can be used to filter sensor values or actuate devices for particular intervals of time. Absolute timing can be used to implement mode changes at different times of the day.

One of our major motivations for using boolean logic is the ability to leverage existing VLSI design tools. Boolean manipulation of sensor-actuator logic can be used to assign blocks of logic to specific physical locations. This is analogous to the synthesis, technology mapping and layout steps used in chip design. Techniques like expression decomposition, extraction, factoring, substitution and term collapsing can be used to minimize massages, improve reliability and load-balance throughput. Logic synthesis verification techniques can identify race conditions, meta-stable states, "stuck-at" faults and cycles all of which should be brought to the

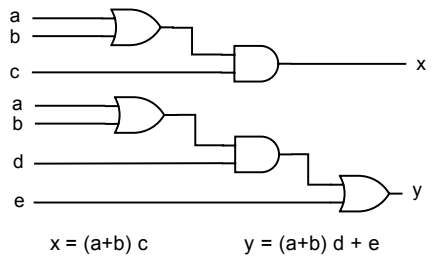


Fig. 2. Simple circuit diagram modeling two independent sensor-actuator configurations. We see the potential for simplification to reduce the common $(a + b)$ term.

attention of the designer before deployment. In our current implementation, we provide SAN-Logic-specific scripts to the SIS [16] sequential logic manipulation package to detect errors, merge multiple expressions and to optimize the amount of message passing. We also demonstrate how common existing programming languages like Relay Ladder Logic and hardware description languages can directly map to SAN-Logic descriptions.

The organization of this paper is as follows. Section II discusses related work. Section III presents the SAN-Logic architecture. Section IV discusses how logic manipulation can be used to optimize and manage systems. Section V addresses the limitations of the design paradigm and Section VI shows a home automation example that illustrates multiplexing of tasks on a single network. Section VII provides concluding remarks.

II. Related Work

Multiple efforts have investigated the streamlining of sensor network programming. Initial systems like TinyDB [15] and Cougar [21] used SQL-like database queries to collect sensor information. Queries can efficiently be used to aggregate and process data as it returns up a tree to the gateway. These approaches are ideal for data collection applications. In [11], the authors presented a middleware service that allows users to specify timing properties associated with requests. Many of the optimizations performed by all these systems are similar to those found in SAN-logic. The largest difference is that the output of SAN-logic is a massively parallel set of asynchronous interactions that execute independently of a central authority. No gateway is required to issue or collect responses from the nodes. We also believe that the logic-based framework simplifies the implementation of certain optimization strategies. The simplicity of dual-state logic optimization is largely responsible for the adoption of binary logic in digital design rather than a larger number of basic states.

Virtual machines provide the basis of a more flexible platform for supporting macro-programming. Maté [10] is a stack-based virtual machine loosely based on the Forth programming language. The VM was designed to support safe and energy-efficient dynamic reprogramming of nodes. The Token Machine Language (TML) [13] is similar to Maté except that executables in TML are compiled down to a binary intermediate form that is executed on each node by a state machine. TML uses an execution and communication model based on token passing where computation is triggered by the arrival of tokens which, in turn, change state stored in shared memory.

Regiment [14], Flask [12] and Semantic Streams [20] are examples of high-level programming languages that allow users to define global operations. These languages use the notion of *stream operators* along with information about location and topological relationships to operate on spatially-defined regions of a network. Similar to SAN-Logic, these approaches define the application specifications independent of the physical network topology. However, they need to be compiled down into an executable form which remains a significant challenge. Regiment was one of the driving motivators for the creation of TML as an intermediate language. In the case of SAN-Logic, the execution engine is much simpler than that of TML or Maté. With SAN-Logic, the core interfaces to sensor and actuators must be implemented as low-level drivers while the rest of the interactions are processed by a logic evaluation engine running on each node. For the same reasons that operating system drivers are implemented close to the kernel, many sensor and actuator drivers require binary implementations for performance reasons. For example, reading and averaging ADC values at a high rate, or decoding pulse width modulation values from a GPIO pin would be difficult if implemented in a virtual machine.

Frameworks like MacroLab [9] and Kairos [8] provide ways to specify high-level applications that are then compiled down into node-specific binary files. Kairos uses a traditional imperative programming model to support general-purpose processing. Parallelizing and deploying imperative programs is notoriously difficult, making it hard to optimize across multiple tasks running in the system. In contrast, MacroLab uses vector programming which can easily be decomposed and mapped across the network. Vector programming requires the user to design parallel algorithms from the start which may prove difficult for non-experts. One goal of SAN-Logic is to provide a work-flow such that field technicians can quickly and easily deploy sensor-actuator interactions in a manner that they are already familiar with.

Multiple researchers have looked at ways to group devices based on various functional or proximity-based properties. These approaches allow devices to be manipulated at the group-level, abstracting away the details of developing each underlying interaction. Hood [19] is a neighborhood programming abstraction that clusters node interactions based on parameters like distance, or the number of wireless hops within the network. Abstract Regions [18] provide a similar mechanism to group nodes based on geographic location or radio connectivity. Nodes can then locally share and process data in order to reduce bandwidth and improve energy performance given a specific accuracy requirement. Bundles [17] are another grouping mechanism that allows management across heterogeneous networks of devices. It provides support for mobility when devices are required to move between various different sub-networks. These approaches tend to focus on defining the communication mechanisms to support distributed applications. In contrast, SAN-Logic uses the data-flow of the application logic to synthesis the communication mapping based on the underlying network topology.

TinyGALS [5] takes a formal concurrency modeling approach to developing distributed embedded software. This approach is probably most similar to SAN-Logic in how it allows for analysis of concurrency at a high-level. Using the galsC [6] language, designers implement particular methods that reside inside each component. These methods are then distributed to each node for execution. Since SAN-Logic operates on Boolean logic statements, it can optimize across multiple independent tasks given a particular instance of a

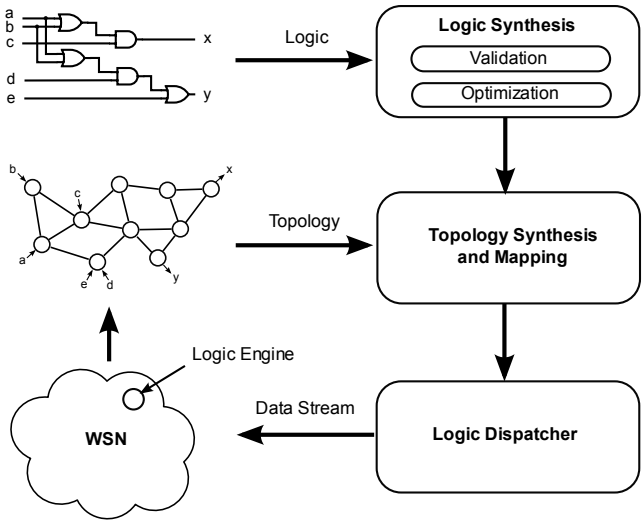


Fig. 3. Design and deployment flow.

network topology to reduce message passing.

Control and automation systems have used Programmable Logic Controllers (PLCs) to manage sensor and actuator control loop for multiple decades. These systems were designed primarily to be extremely simple to configure and manage at a large scale. PLCs are typically programmed using graphical languages that emphasize logical organization of operations. The most popular example of this is Relay Ladder Logic (RLL). RLL shares SAN-Logic's notion of modeling the system as hardware components. In fact, we show in Section III-B that RLL can be directly converted into boolean expressions making it a viable programming front-end for SAN-Logic. RLL typically represents a set of rules that are sequentially evaluated by a cyclic executive. This loop is rapidly executed such that interactions seem nearly instantaneous. Unfortunately, this can lead to race conditions and oscillating states. SAN-Logic attempts to remedy these drawbacks by providing timing properties associated with each signal. Unlike RLL, SAN-Logic is inherently designed to run on multiple nodes in a mesh network with inter-node message passing.

The field of VLSI design has revolved around designing, optimizing and synthesizing systems described by boolean expressions. This would not be possible without the seminal work related to compact logic representation [4], logic optimization [3] and synthesis [2]. Many researchers have studied automata theory and finite state machines that result from the incorporation of latches in digital circuits. SAN-Logic provides a runtime environment and the key technology mapping primitives to apply these techniques to wireless sensor networks. In our implementation we use the SIS [16] sequential logic package that is built around a large body of research and development.

III. Architecture

Figure 3 shows a diagram that outlines the high-level design and deployment flow of SAN-Logic. Initially, a logic description of the set of interactions is passed to a *Logic Synthesis* sub-system. This sub-system optimizes and validates the input logic without using topology information. Common expressions can be combined and errors like race conditions can be identified. This optimized logic is then passed to a topology-mapping stage where logic is again synthesized but is used in conjunction with topology information. Now, it is possible to optimize for different QoS metrics by manipulating and duplicating routes. Ideally, the topology information should be collected directly from the sensor network. Finally, the system description is packetized into individual components that need to run on each node and dispatched to them over the network. Each node runs a local *Logic Engine* that is responsible for evaluating and reacting to inputs.

In the remainder of this section we will describe the individual components that comprise SAN-Logic. First, we outline the primitives used to define the language. Next, we discuss various methods that a designer can use to model the system. This includes a description of current PLC approaches and how they can be mapped to run using SAN-Logic on a sensor network. We then discuss mechanisms that the synthesis step can use to optimize the network and describe existing verification techniques which can apply to sensor-actuator systems. Finally, we describe the runtime environment that executes on each sensor node.

A. Language Primitives

A SAN-Logic system is described by boolean variables related to each other using a combination of AND (*), OR (+) and NOT (!) logical operators. The language supports timed asynchronous sequential logic [1]. Sequential logic, as opposed to combinational logic, can feedback outputs from logical expressions to capture state machines. *Signals* are used as a means to convey variable values from one expression to another. A signal combines the current value of a variable with timing parameters that govern its transmission. Even though the transmission of signals can be delayed, upon reception, they are processed immediately and hence exhibit an asynchronous quality. There are three types of signals: sensor signals, actuator signals, and intermediate signals. Sensor and actuator signals generate the input and output of the system and hence require a description of physical capabilities. This includes sensor/actuator type, the MAC address of the physical node and a description of how the transducer should process the input or output data. In our logic specification, variables are defined in the following format:

```
#define variable_name {actuator|sensor|int}
   < MAC address, type, operator, [params ] >
```

The *type* of the sensor indicates which existing low-level driver should be called to generate the signal. This requires parameters that define attributes like the thresholding mechanisms, threshold types, sampling rates, etc. Below is a typical description of a sensor and an actuator:

```
define x actuator <0x4a,AC_PWR,sock0>
define a sensor <0x03,LIGHT,gt,100>
```

A variable x is defined turn on or off socket 0 of an AC outlet on node *0x4a*. Variable a is defined as a light sensor on node *0x03* that becomes `TRUE` when its value is greater than 100. Sensor operators (like `gt`) are eventually translated into numerical indexes used by low-level drivers. These drivers are device-specific, but typically support operations like: *gt* (greater than), *lt* (less than), *eq* (equal), *dfi* (delta from initial value), *dfl* (delta from last sample) etc. Unspecified variables are defined by default to be intermediate signals which become messages that get passed within the network. Intermediate signals can also be defined with timing parameters. In cases where intermediate signals are used to form sequential logic statements (those with feedback), initial startup values must be supplied.

Timing parameters and the logical expressions can be represented in assignment form. The simplified left-hand term is the signal generated by the signals or variables on the right-hand side of the equation. Timing properties for the signal are captured in the curly braces following each statement. Each signal defined in SAN-Logic can have different timing and queuing parameters for *rising-edge* (`delay_re`) or *falling-edge* (`delay_fe`) state transitions. Currently, the logic engine supports the following queueing policies: *FIFO*, *replace*, and *cancel*. *FIFO* maintains a queue with first-in first-out ordering with a time delay specified by the delay parameter. *replace* will substitute new items in place of existing items with the same signal name. Any existing delay parameter will also be overwritten. *cancel* will remove a signal from the queue if an opposite value is inserted before the initial value has been transmitted. These operations simplify filtering inputs and enable the creation of timed state machines. If a motion detector is configured to turn on a light, we would ideally like to keep the light active until the motion detector is idle for an extended period of time rather than the first time it fails to detect motion. Below is an example that illustrates this principle:

```
x = a + !b {delay_re: 0ms, policy_re: fifo,
       delay_fe: 10000ms, policy_fe: cancel}
```

In this case, we have configured signal x to be dependent on the value of a or the inverse of b. The rising-edge delay is set to 0ms with a FIFO policy. This means that when the inputs produce a `TRUE` signal, the message is sent immediately. The falling-edge is governed by a *cancel* policy which means that if the signal changes within the timeout period of 10 seconds, the `FALSE` signal will be deleted. This means that the inputs must be stable for more than 10 seconds before the output can go from high to low. A {`time: type value`} parameter delays a signal until an absolute wallclock time. This can be specified in seconds, minutes, hours, days, weeks, months and/or years. For example, an action can be specified for every day at 5pm with the following time parameter: {`time: hour 17`} or to start each Saturday with {`time: day Saturday`}.

In order to describe non-binary data, variables can be grouped using bit concatenation forming buses. We adopt the hardware description language syntax of using square braces $[x : y]$, which denotes bits y through x of the bus. For example, in the follow expression:

```
x[4:0] = y[4:0] * b
```

x is a 4-bit value that can either be set to the value produced by the 4-bit value of y or not based on the value of b.

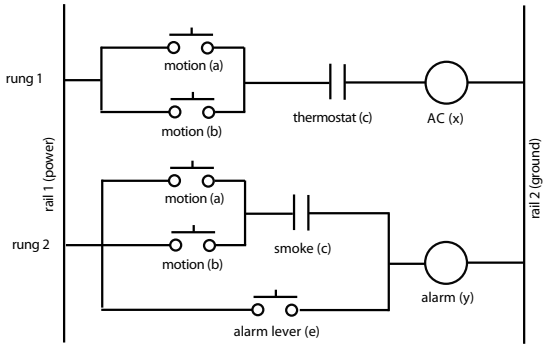


Fig. 4. Relay Ladder Logic representation of circuit shown in Figure 2.

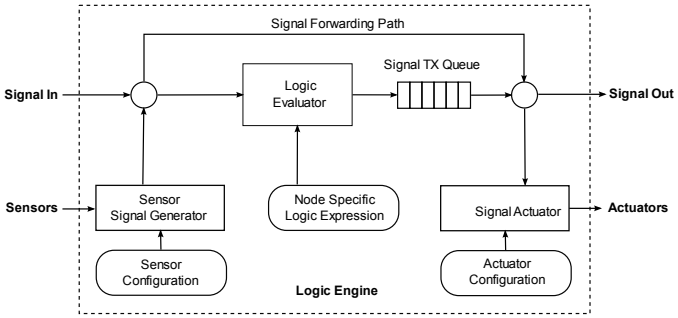


Fig. 5. Logic Engine running on node.

B. Logic Designer

As described above, one possible method of describing a system is through a text description of sensors, actuators and their interdependencies. However, to achieve our goal of a truly user-friendly design framework, a graphical interface is desirable. One option is to design a layout tool that provides a gate-level drawing of circuits. This approach is commonly used in industrial automation to configure PLCs using Relay Ladder Logic (RLL). RLL is a graphical diagram-based language that connects components like elements in a circuit that can be captured as a subset of SAN-Logic. Figure 4 shows an example ladder logic program that describes the same interactions shown in Figure 1 and Figure 2. The vertical rail on the left side of the figure represents power that flows to a ground rail on the right side of the figure. Each horizontal rung represents a logical expression and can be captured using boolean algebra. In RLL, the rungs are executed sequentially such that any value computed in a higher rung is available in lower rungs. Ladder logic can include delay blocks, but capturing intricate timing properties remains quite difficult. Never-the-less, ladder logic is a simple initial interface for modeling many sensor-actuator networks and is quite popular.

Other possible approaches include using hardware description languages like VHDL and Verilog. Procedural and behavioral descriptions that can be synthesized into structural components can be directly mapped to a SAN-Logic system description.

C. Logic Synthesizer

There are two primary logic synthesis steps that are used to optimize designs for particular QoS properties before deployment. The first is a logic synthesis step that operates on the input logic without any notion of topology. During this stage, the boolean expressions are manipulated to factor out common terms and reduce the number of literals and operators in an expression. Next, a synthesis step occurs that uses topology information collected from the sensor network. At this point, additional boolean operations can be performed, but the primary challenge is associated with mapping signals to the topology. We discuss possible QoS metrics of interest and detail various optimization techniques in Section IV.

D. Verification

Verification and error checking are critical when designing complex distributed systems that interact with the environment. In this section, we briefly summarize the most common model checking techniques which exist in logic synthesis tools that can be used to analyze SAN-Logic systems. One of the most fundamental verification requirements is the ability to determine if one circuit is functionally identical to another which is called equivalence checking. Equivalence checking can be used to perform sanity checks on optimized components, or to compare functionality of a circuit with that of a reference design. Though out of the scope of this paper, techniques have been developed for both combinational as well as sequential equivalence checking in the area of formal methods. The use of Binary Decision Diagrams [4] (BDDs) provides a compact and efficient representation of binary logic that aids in many of these algorithms. When a full representation of the desired system is not available, test vector simulation can be used to verify a subset of output values. Simulation of SAN-Logic without timing delays is possible in most logic synthesizer packages. Minor adjustments would be required to simulate the system given different transmit queue policies. Reachability analysis can be used to determine if all states in a sequential logic system can be reached. Unreachable states likely indicate design errors and should be brought to the user's attention. So-called *stuck-at* faults, where an output is always fixed at a particular value, irrespective of any input should also be flagged as errors. Along these same lines, a designer might inadvertently create a logical cycle that continuously oscillates an output. These situations can occur by design, however, they should be flagged as warnings since they are often unintended.

E. Logic Engine

Figure 5 shows a block diagram of the *Logic Engine* that executes on each sensor-actuator node. The logic engine is responsible for arbitrating all input and output signals based on the given logic expressions and sensor-actuator configurations.

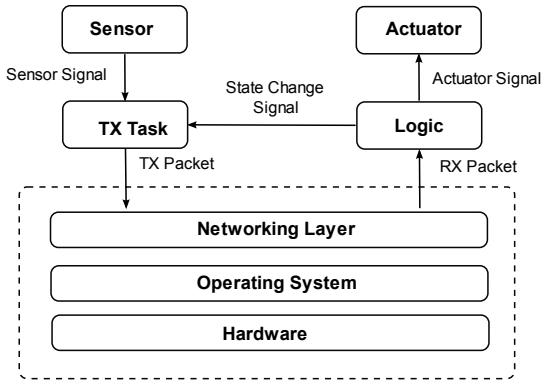


Fig. 6. Tasks executing on node.

Signals received as packets by the node are passed to the logic engine and in turn any generated signals from the logic engine are passed to the network. The Logic Engine will only transmit signals if their state has changed, or if there is an explicit request for state updates from neighbors. The logic expressions for each node are stored as Reverse Polish Notation (RPN) expressions that are created by the dispatcher and addressed to each specific node in the network. The signal TX queue is used to store signals in accordance with their timing parameters. Figure 6 shows a breakdown of tasks that comprise the logic engine. Our implementation of SAN-Logic was designed to run on top of the Nano-RK [7] real-time operating system. Nano-RK is a fully preemptive multitasking OS, which allows each rectangle in Figure 6 to be designed as a separate concurrent task. The dotted rectangle indicates the components that are part of Nano-RK that simply support the logic engine. The total compiled SAN-Logic executable consumes 30K of flash and 2.6K of RAM when compiled for FireFly sensor nodes. This is quite compact given that 26K of flash and 1.2K of RAM are consumed by the OS and underlying network layer.

IV. Network Optimizations

In this section, we describe how the manipulation of expressions using boolean algebra can be used to adjust network performance. There are many dimensions of QoS that could be optimized in a sensor-actuator network based on the application including energy, throughput, message latency, reliability

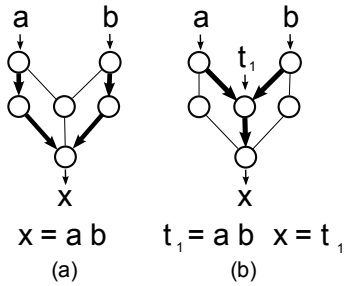


Fig. 7. Adding terms and mapping them to nodes can change message passing behavior.

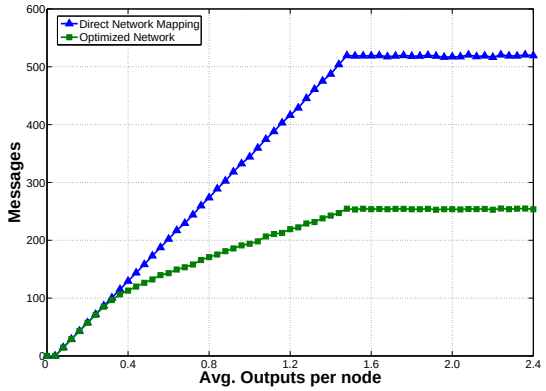


Fig. 8. Number of messages as a function of average number of output terms mapped onto each node. As the number of outputs increase, the opportunity to share input data increases.

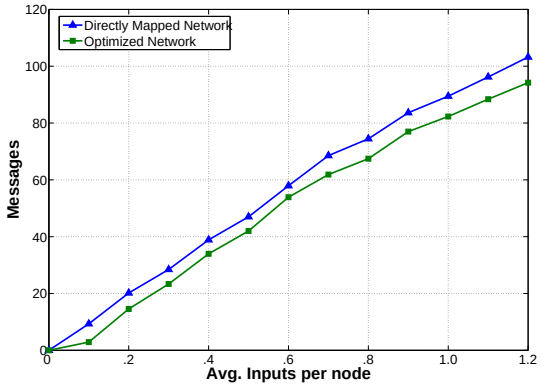


Fig. 9. Number of messages as a function of average number of input terms mapped onto each node. As the number of inputs increase, the opportunity to optimize data remains relatively consistent.

and extensibility. We focus primarily on energy reduction. However similar techniques can be used to optimize for any of the other metrics. We show that by combining common terms and intelligently mapping them, we can significantly reduce the number of messages required to react to changing inputs

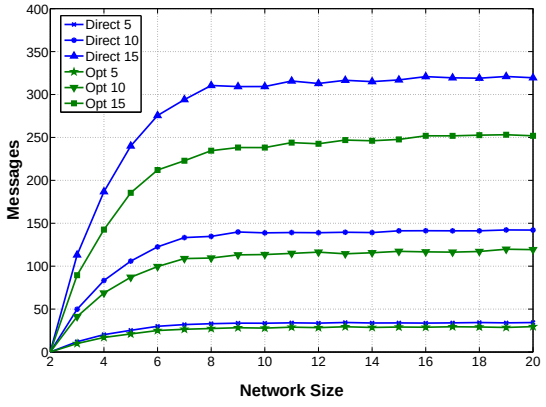


Fig. 10. Number of messages as a function of network size for various numbers of inputs and outputs.

in the system.

A. Topology Mapping

When mapping a boolean expression onto a network, one approach is to pass all relevant sensor inputs to their associated actuator output nodes which in turn locally evaluate the logic expressions for each output. This is analogous to what human designers often do when building sensor-actuator systems that directly pass messages from sensors to their destination actuators. This approach is functionally correct; however, it does not capitalize on the ability to manipulate the underlying logic expressions which may contain common terms. Input and output signals generated by transducers must be mapped to their corresponding physical node. Intermediate signals can be mapped anywhere on the network topology if it is a connected graph. Additional terms can be added to expressions that do not change their functionality, but simply provide a means of moving message passing patterns around in the network. Take, for example, the expression $x = ab$, where x is an actuator while a and b are sensors. Using shortest path routing, one possible configuration is shown in Figure 7 (a). By adding an intermediate term and mapping that to the topology, we can change the way messages are passed as shown in (b) with the addition of signal t_1. In the next section, we will show how this simple concept in conjunction with logic simplification techniques can provide a powerful means to reduce messages count which saves energy.

B. Literal reduction and short-circuited expressions

The goal of most logic simplification tools is to reduce the overall literal count of a logic expression usually resulting in fewer gates. All logic expressions can be represented as a two-level Sum-Of-Products (SOP) where each input is AND'ed together forming intermediate values that are passed to a single OR gate. The problem with this approach is that many AND gates are needed to encode the entire truth-table. Logic synthesis techniques like Karnaugh maps, Quine-McCluskey and Espresso [2] convert SOP equations into smaller multi-level logic functions. By increasing the depth of the logic, they capitalize on reusing common terms in order to minimize gates. This same concept can be used to reduce the number signal literals in a SAN-Logic scheme. Unfortunately, logic simplification is known to be an NP-hard problem. Karnaugh maps and the Quine-McCluskey algorithms require exponential time and are not suitable for expressions beyond a few variables. Espresso is based on heuristics and can be quite effective at minimizing literals.

Figure 7 (a) shows a topology where messages are passed from sensors to actuators using the naive shortest path approach where all execution occurs at the actuator. Notice that there are two sets of expressions that share the common term ab. Figure 7 (b) shows the logic output after running Espresso. The obvious simplification is to combine the ab terms. However, care must be taken when placing the intermediate variables. If the intermediate signal is placed at an extreme location, near the x term, then the overall number of

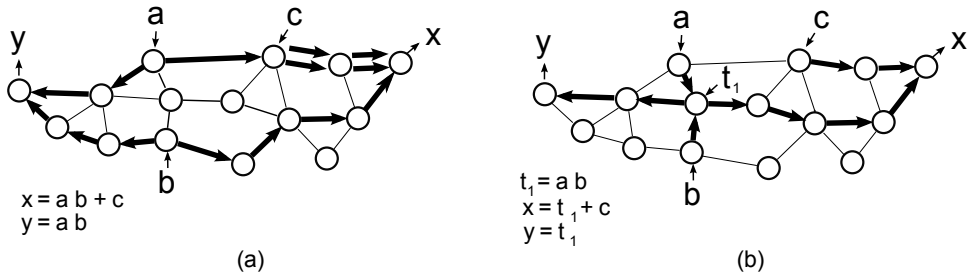


Fig. 11. Combining terms across expressions can significantly reduce message passing. Often times there is the additional benefit of terms being evaluated near the inputs (short-circuit evaluation) with no change which suppresses further message passing.

messages would increase. We apply a heuristic that places the intermediate terms at the node in the graph that minimizes the distance from all input values. The reasoning for biasing the location towards the inputs and not the outputs is because many times if the expression is locally evaluated and does not change, additional messages can be avoided. This is known as *short-circuit* evaluation which is a property of boolean operators, where the outcome of an early operation removes the need to perform further evaluation. As with most heuristics, there are situations where this is not optimal. However, in practice, it performs quite well.

In order to evaluate the effectiveness of different optimizations, we require a method for comparing the performance of two given topologies with logic interactions. Ideally, we would need to simulate the number of message transactions given every possible set of input transitions. This approach is computationally infeasible even for medium-sized networks and assumes a uniformly distributed workload model which may change based on the application. Instead, we approximate this value by calculating the number of message transactions between each related signal across the network. This is achieved by summing up the total messages required for each right-hand term to reach the left-hand destination across all expressions. For example, in Figure 11 (a), the number of messages is computed as follows:

$$x = ab + c \; (a \rightarrow x = 3)(b \rightarrow x = 4)(c \rightarrow x = 2) \quad (1)$$

$$y = ab \; (a \rightarrow y = 2)(b \rightarrow y = 3) \quad (2)$$

$$\sum x + \sum y = 14 \quad (3)$$

For Figure 11 (b) the total number of messages is:

$$t_1 = ab \; (a \rightarrow t_1 = 1)(b \rightarrow t_1 = 1) \quad (4)$$

$$x = t_1 + c \; (t_1 \rightarrow x = 4)(t_1 \rightarrow y = 2) \quad (5)$$

$$y = t_1(t_1 \rightarrow y = 2) \quad (6)$$

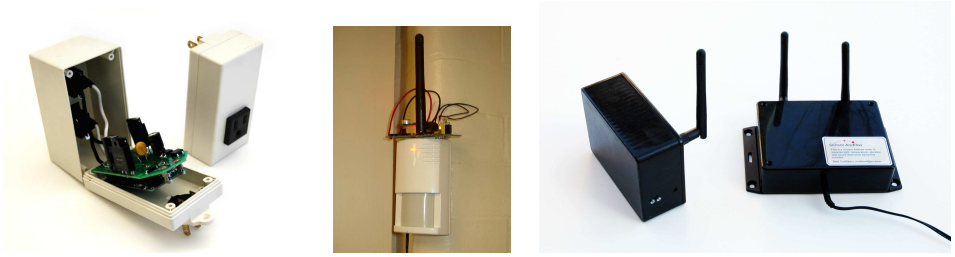
$$\sum t_1 + \sum x + \sum y = 10 \quad (7)$$

Using the above metric, we evaluate how optimizing the logic given different sensor and actuator configurations perform across networks at different scales. In the following experiments, we randomly generate logic expressions in SOP form and uniformly mapped inputs and outputs onto a randomly generated geometric graph. Figure 8 shows that our optimizations drastically decrease the number of messages as the number of output terms increase. At above 1.6 outputs per node we see a consistant reduction in messaging of about 40%. The intuition behind this is that as the number of outputs increases, the amount of shared information between outputs also increases and hence factoring out common terms and placing them in central locations will decrease overall message passing. After each node has on average more than one output per node, we see the number of messages begin to saturate since each node is already communicating with all other nodes in the network. In contrast, in Figure 9, we see that as the number of inputs increase and the outputs remain largely static, the optimization is less effective. With a large number of uniformly distributed inputs, finding central locations to combine common factors becomes increasingly difficult. Generally, they all begin to converge at the center of the graph and then fan out towards the outputs. It is still worth noting that optimization never performs worse and does on average 5-10% better. Figure 10 shows three different sets of input, output and expression sizes with respect to number of nodes in the network. Direct-X and Opt-X in the legend refers to the direct mapping approach for input, output and expression size X in comparison with the optimized logic mapping. We generally see that as the system complexity increases, optimizing the logic further decreases the proportional number of messages as compared to the direct mapping method.

It is possible to further optimize the placement of signals on the graph by attempting to simplify each expression into a minimum spanning tree. Figure 13 shows a variant of the previous test topology using this strategy. We see that the original message count of 14 is now reduced to 7. In practice, this approach may not perform as well as our original heuristic if the system tends to be biased towards a particular set of inputs. In this example, if b changes state more frequently than a, there would be more messages generated than in our previous example.

C. Coping with packet-loss

In any real sensor networking system one faces packet-loss as well as intermittent node failures. These failures could be due to actual hardware failures, or could be as rudimentary as an object temporarily blocking a node's antenna. We adopt the standard approach of using link-level acknowledgments and retries to increase reliability. Typically a device only transmits



(a) (b) (c)

Fig. 14. Various transducers used in our home automation application (a) FireFly wall power outlet sensing and actuation device (b) PIR and microwave motion detector interfaced with a FireFly sensor node (c) FireFly sensor node equipped with light, temperature, audio, humidity and acceleration sensors sitting next to an FireFly to 802.11 gateway device.

when there is a possible state change. To increase reliability, nodes will also periodically rebroadcast state information. This periodic rebroadcast can be adjusted at runtime based on link metrics with neighboring nodes. In our implementation, we used the average transmit retry at each node to set the gain for adjusting the periodic node update rate. Nodes with low retry counts would rarely send out periodic updates, while node with higher retry rates would more frequently send messages. This approach can be used in conjunction with multi-path routing to operate in the presence of node failures.

V. Limitations

SAN-Logic is best suited for systems with binary variable values. Non-binary values can be passed using the signal bus construct that allows multiple signals to represent individual bits of a variable. Extensive use of wide buses will incur significant synthesis time overhead since each bus is internally represented as a set of individual signals. Wide buses will also increase the size of the logic descriptions passed dispatched to nodes. SAN-Logic is generally poorly suited for modeling control interactions based on continuous functions and those that depend heavily on complex state machines. Sequential logic optimization systems suffer from scaling due to issues of NP completeness and state space explosion problems. Current FPGA systems support millions of gates, which indicates that the tools involved in logic synthesis should be able to support even the most demanding sensor network interactions that we envision in the near future.

VI. Example Application

In this section, we describe a home automation application where the users specifies multiple sets of interactions using sensors and actuators in a home environment. We outfitted an apartment with the four types of wireless sensor networking transducer shown in Figure 14. This included nine wall-plug power transducers, three motion detectors, five environmental sensor nodes and an 802.11 gateway. Each transducer has a FireFly sensor node which uses a cc2420 radio and an ATmega1281 micro-controller. The nodes run SAN-Logic implemented on the Nano-RK real-time operating system. The wall-plug power transducer shown in Figure 14(a) can not only sense the amount of power that connected appliances are consuming, but it can also enable or disable the outlet with commands from the wireless network. Three lights, located in different rooms, were independently controlled by the power nodes. The power usage from the TV and a lamp are combined to determine if the overhead lamp in the living room should be disabled while watching TV. Figure 14(b) shows a combination passive infrared and microwave motion detector that is wired to a FireFly sensor node. These nodes run a SAN-Logic driver capable of generating 1 second signal pulses when motion is detected. Each FireFly environmental sensor, shown to the left in Figure 14(c), generates signals based on thresholded analog inputs from temperature, light, sound-level and acceleration sensors. The FireFly gateway provides a link to the WiFi network inside the apartment so that the network can be securely configured and remotely monitored.

The textual description of the system can be found in

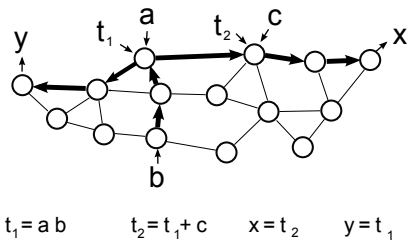


Fig. 12. Further reduction is possible by performing spanning tree optimizations ontop of the logic.

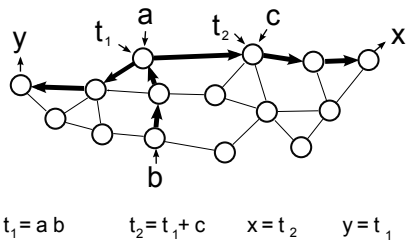


Fig. 13. Further reduction is possible by performing spanning tree optimizations ontop of the logic.

```
#define motion_1    sensor   <0x1,binary>
#define motion_2    sensor   <0x2,binary>
#define motion_3    sensor   <0x3,binary>
#define ff_light_1  sensor   <0x4,LIGHT, gt, 200>
#define ff_light_2  sensor   <0x5,LIGHT, gt, 200>
#define ff_light_3  sensor   <0x6,LIGHT, gt, 200>
#define ff_sound_1  sensor   <0x4,AUDIO, dfl, 10>
#define ff_sound_2  sensor   <0x5,AUDIO, dfl, 10>
#define ff_sound_3  sensor   <0x6,AUDIO, dfl, 10>
#define ff_temp_1   sensor   <0x4,TEMP, gt, temp_set_point>
#define plug_node_1 actuator <0x7,AC_PWR, sock_all>
#define plug_node_2 actuator <0x8,AC_PWR, sock_all>
#define plug_node_3 actuator <0x9,AC_PWR, sock_all>
#define plug_node_4 actuator <0xa,AC_PWR, sock_all>
#define light_1     actuator <0xb,AC_PWR, sock_0>
#define light_2     actuator <0xb,AC_PWR, sock_1>
#define light_3     actuator <0xc,AC_PWR, sock_2>
#define tv_pwr     sensor   <0x7,AC_PWR, sock_0>
#define light4_pwr  sensor   <0x7,AC_PWR, sock_1>
#define set_point[8]   int   <0x4,TEMP,INT>
#define ac          actuator <0xe,COOLING_ON>


mot_occupied = motion_1 + motion_2 + motion_3
                   {delay_re: 0ms, policy_re: fifo,
                    delay_fe: 30000ms, policy_fe: cancel}

audio_occupied = ff_sound_1 + ff_sound_2 + ff_sound_3
                   {delay_re: 0ms, policy_re: fifo,
                    delay_fe: 10000ms, policy_fe: cancel}

occupied = audio_occupied + mot_occupied

% Show an array clustering inputs
all_off[0] = plug_node_1
all_off[1] = plug_node_2
all_off[2] = plug_node_3
all_off[3] = plug_node_4
all_off[4] = plug_node_5

% disable lights when unoccupied
all_off = !occupied

light_1 = !(motion_1 + ff_light_1)
                   {delay_re: 0ms, policy_re: fifo,
                    delay_fe: 30000ms, policy_fe: cancel}
light_2 = !(motion_2 + ff_light_2)
                   {delay_re: 0ms, policy_re: fifo,
                    delay_fe: 30000ms, policy_fe: cancel}
light_3 = !(motion_3 + ff_light_3)
                   {delay_re: 0ms, policy_re: fifo,
                    delay_fe: 30000ms, policy_fe: cancel}

% Mood Lighting Scenario
% Also control light 1 based on TV and light 4 usage
light_1 = (tv_pwr * light4_pwr) * occupied
                   {delay_re: 0ms, policy_re: fifo,
                    delay_fe: 30000ms, policy_fe: cancel}

% turn on AC if temperature is too hot for 120 seconds
% when the apartment is occupied
ac = ff_temp_1 * occupied
                   {delay_re: 0ms, policy_re: fifo,
                    delay_fe: 120s, policy_fe: cancel}
```

Fig. 15. Text description of home automation application.

Figure 15 with its corresponding graphical description shown Figure 16. In this example there are three distinct operation configured by the user that are being mapped onto the system. The first is occupancy-based management of lights. When the user is not present in the apartment then the lights are configured to be disabled. The second task is management of lights based on TV usage as well as other lighting conditions in the room. The third task is control of an air conditioning window unit based on occupancy and temperature from one of the nodes. In this example we see how a mixture of

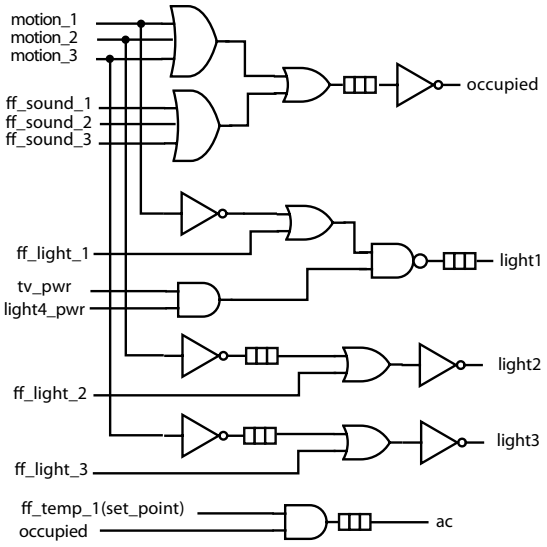


Fig. 16. Home automation application circuit.

sensors and actuators could benefit from being shared across different tasks. Using a structured approach like SAN-logic, these interactions can also be optimized to improve networking performance in distributed manner. Reduction of messages serves two purposes: (1) it increases lifetime for battery operated nodes and (2) it helps reduce traffic on the network which limits scalability and adds to timing jitter. In our network, the battery operated nodes spend the majority of thier energy on communication (98%). After running our logic reduction operations on this network we saw 15% less messages due to early signal termination (short-circuit operations). As shown in the experimental evaluation section, this performance would likely further improve in larger networks like those found in commercial buildings.

VII. Conclusion

This paper presented a logic-based programming paradigm for wireless sensor-actuator networks designed for CPS applications called SAN-Logic. SAN-Logic provides the following main CPS programming features: (1) specification of both sensors and actuators, (2) incorporation of physical topology information, (3) support for timing requirements and (4) support for heterogeneous devices. Our approach uses sequential boolean logic relations between inputs and outputs to model interactions in the network. Modeling the system using boolean logic allows us to leverage VLSI hardware design approaches to simplify management and deployment of large-scale networks. We provide techniques using these existing logic optimization packages that combine common terms and benefit from short-circuit evaluation of expressions to reduce message passing. In complex systems we see up to a 40% reduction in number of messages required to respond to changing inputs as well as a 15% reduction in our apartment deployment. Using the VLSI tool-chain allows SAN-Logic systems to be designed in existing languages like Relay

Ladder Logic and hardware description languages like VHDL and Verilog. We enhance the boolean logic representation of the system by including rich timing properties making the language more suitable for real-time interactions with the environment required by CPS applications. Each transition of a signal, rising or falling, can be assigned a delay value and policy based on a relative or absolute time. This allows for filtering of spurious sensor values and accurate timing of actuation events.

REFERENCES

[1] Ashar, P., Devadas, S., Newton A.R. Sequential logic synthesis. *Springer*, 1992.

[2] Brayton, R. K., Hachtel, G. D., Curtis, T., Sangiovanni-Vincentelli, A. Logic Minimization Algorithms for VLSI Synthesis. *Kluwer Academic Publishers*, 1984.

[3] Brayton, R. K., Rudell, T., Sangiovanni-Vincentelli, A., Wang, A.R. MIS: A Multiple-Level Logic Optimization System. *IEEE Computer-Aided Design of Integrated Circuits and Systems*, 1987.

[4] Bryant, R. Graph-based algorithms for Boolean function manipulation. *IEEE Transaction on Computers*, 1986.

[5] Cheong, E., Liebman, J., Liu, J., Zhao, F. TinyGALS: A Programming Model for Event-Driven Embedded Systems. *ACM Symposium on Applied Computing*, 2003.

[6] Cheong, E., Liu, J. galsC: A Language for Event-Driven Embedded Systems. *Proceedings of the conference on Design, Automation and Test in Europe (DATE)*.

[7] Eswaran, A., Rowe, A., Rajkumar, R. Nano-RK: an Energy-aware Resource-centric RTOS for Sensor Networks. *IEEE Real-Time Systems Symposium (RTSS)*, 2005.

[8] Gummadi, R., Gnawali, O., Govindan, R. Macro-programming wireless sensor networks using Kairos. *Intl Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2005.

[9] Hnat, T., Sookoor, T., Hooimeijer, P., Weimer, W., Whitehouse, K. MacroLab: A Vector-based Macroprogramming Framework for Cyber-Physical Systems. *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2008.

[10] Levis, P.,Culler, D. Mate: a tiny virtual machine for sensor networks. *International Converence on Architectural support for programming languages and operating systems (ASPLOS)*, 2002.

[11] Li, S., Lin, Y., Son, S. H., Stankovic, J., Wei, Y. Event detection services using data service middleware in distributed sensor networks. *ACM International Conference on Information Processing in Sensor Networks (IPSN)*, 2003.

[12] Mainland, G., Morrisett, G., Welsh, M. Flask: Staged Functional Programming for Sensor Networks. *International Conference on Functional Programming (ICFP)*, 2008.

[13] Newton, R., Arvind, Welsh, M. Building up to Macroprogramming: An Intermediate Language for Sensor Networks. *International Converence on Information Processing in Sensor Networks (IPSN)*, 2005.

[14] Newton, R., Morrisett, G. and Welsh, M. The Regiment Macroprogramming System. *International Conference on Information Processing in Sensor Networks (IPSN)*, 2007.

[15] S. Madden, M. J. Franklin, J. M. Hellerstein and W. Hong. TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks. *Operating Systems Design and Implementation (OSDI)*, 2002.

[16] Sentovich, E., Singh, K., Moon, C., Savof, H., Brayton, R., Sangiovanni-Vincentelli, A. Sequential circuit design using synthesis and optimization. *Computer Design: VLSI in Computers and Processors (ICCD)*, 1992.

[17] Hoque E. Stankovic J. Xie Z. Vicaire, P. Bundle: A group based programming abstraction for cyber physical systems. *IEEE/ACM Conference on Cyber-Physical Systems (ICCPS)*, 2010.

[18] Welsh, M., Mainland, G. Programming sensor networks using abstract regions. 2004.

[19] Whitehouse, K., Sharp, C., Brewer, E., Culler, D. Hood: A Neighborhood Abstraction for Sensor Networks. *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2004.

[20] Whitehouse, K., Zhao, F., Liu, J. Poster Abstract: Automatic Programming with Semantic Streams. *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2005.

[21] Yao Y., Gehrke J. The cougar approach to in-network query processing in sensor networks. *ACM Special Interest Group on Management of Data (SIGMOD)*, 2002.