galsC: A Language for Event-Driven Embedded Systems

Elaine Cheong
Department of EECS
University of California, Berkeley
Berkeley, CA 94720
celaine@eecs.berkeley.edu

Jie Liu
Microsoft Research
One Microsoft Way
Redmond, WA 98052
liuj@microsoft.com

***Abstract*— We introduce galsC, a language designed for programming event-driven embedded systems such as sensor networks. galsC implements the TinyGALS programming model. At the local level, software components are linked via synchronous method calls to form actors. At the global level, actors communicate with each other asynchronously via message passing, which separates the flow of control between actors. A complementary model called TinyGUYS is a guarded yet synchronous model designed to allow thread-safe sharing of global state between actors via parameters without explicitly passing messages. The galsC compiler extends the nesC compiler, which allows for better type checking and code generation. Having a well-structured concurrency model at the application level greatly reduces the risk of concurrency errors, such as deadlock and race conditions. The galsC language is implemented on the Berkeley motes and is compatible with the TinyOS/nesC component library. We use a multi-hop wireless sensor network as an example to illustrate the effectiveness of the language.**

I. Introduction

Networked embedded software designers face issues such as managing computation as well as communications, maintaining consistent state across multiple tasks, handling irregular interrupts, avoiding concurrency errors, and conserving power. These tasks become more challenging when the resources of the hardware platforms are too limited, in terms of CPU speed and memory size, to host a full-scale modern operating system. Traditional technologies for developing embedded software, inherited from writing device drivers and from optimizing assembly code to achieve a fast response and a small memory footprint, do not scale with the growing complexity of today's applications. Despite the fact that "high-level" languages such as C and C++ have recently replaced assembly language as the dominant embedded software programming languages, most of these high-level languages are designed for writing sequential programs to run on an operating system and fail to handle concurrency intrinsically.

Event-driven embedded software is more like hardware, where conceptually concurrent components are activated by incoming signals (or events). Event-driven execution is particularly suitable for untethered devices such as sensor network nodes, since the node can be put into a sleep mode to preserve energy when no interesting events are happening. For many networked embedded systems, there is a fundamental gap between this event-driven execution model and sequential programming languages.

The *TinyGALS* (Globally Asynchronous and Locally Synchronous) programming model [1] aims to fill this gap by providing language constructs to systematically build concurrent tasks (called *actors*). At the application level, actors communicate with each other asynchronously via message passing. Within each actor, *components* communicate synchronously via method calls, as in most imperative languages. Thus, the programming model is globally asynchronous and locally synchronous in terms of transfer of the flow of control. In order to incorporate shared variable semantics where only the latest value matters, a set of guarded yet synchronous variables (called *TinyGUYS*) is provided at the system level for actors to exchange global information "lazily." Access to these variables is thread-safe, yet components can quickly read their values. In this programming model, application developers have precise control over the concurrency in the system, and they can develop software components without the burden of thinking about multiple threads.

In this paper, we introduce *galsC*, a language that implements the TinyGALS programming model. This new approach is based on the nesC language and compiler [2], and improves on the design in [1] in the following aspects:

- A real compiler backend allows the galsC compiler to perform better type checking and code generation.
- A novel global variable (TinyGUYS) handling mechanism does not require special parameter access keywords. This makes galsC compatible with all nesC components.
- This paper also contains a deeper discussion on concurrency features of galsC, in comparison to [1].

The design of galsC is influenced by the trend of introducing formal concurrency models in embedded software. In particular, synchronous languages try to compile away concurrency executions based on the synchronous (zero-execution time) assumption [3]. When it is not possible to compile away concurrency, the port-based object (PBO) model [4] has a global shared variable space mediating component interaction. Various dataflow models [5] use FIFO queues to separate flow of control. The POLIS co-design approach [6] uses an event-driven model for both hardware and software executions. To some extent, galsC is closer to system-level hardware/software codesign languages, such as SystemC [7] and VCC [8], than embedded software languages such as nesC. galsC provides basic concurrency constructs and generates executable code, including an application-specific operating system scheduler, from high-level specifications. This generative approach allows further analysis of concurrency problems, such as race conditions, at a high level. Automatically generated code also reduces implementation and debugging time, since the developer does not need to reimplement standard constructs (e.g. communication ports, queues, functions, and guards on variables).

The remainder of this paper is organized as follows. Section II describes galsC syntax and semantics, connection model, type checking, and code generation. Section III discusses concurrency issues in galsC programs. Section IV describes a sample application

*This work was supported in part by an Intel Open Collaborative Research Fellowship, PARC (Palo Alto Research Center), and UC Berkeley CHESS (Center for Hybrid and Embedded Software Systems). We would like to thank David Gay (Intel Research Berkeley) for nesC, enabling our extension to the nesC compiler, and contributions to the connection model; Edward A. Lee and members of the Ptolemy Project (UC Berkeley) for discussions on the concurrency model; and Feng Zhao for support at PARC.

implemented in galsC. We conclude with directions for future work.

II. The galsC Language

In this section, we overview the galsC syntax and semantics with a simple sensing application (Figure 1). In this example, a hardware clock triggers the system to update a time tick counter. A downsampled clock signal triggers the system to read the light intensity level from a photoresistor at a lower rate. Reading the sensor may take time. The system tags the resulting sensor value with the latest value of the counter and sends it downstream for further processing.

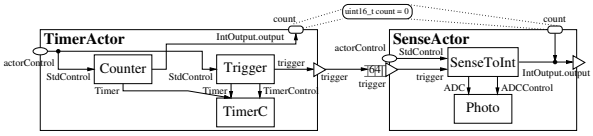

Fig. 1. The SenseTag application.

A. Language constructs

There are three basic constructs in galsC: components, actors, and applications. *Components* are the most basic elements of a galsC program and are written in the nesC programming language. Components provide and/or require *interfaces*, which are collections of methods. A component that provides an interface contains an implementation of the interface method(s), whereas a component that requires an interface expects another component to implement the interface. A component is either a *module* or a *configuration*. A module contains executed code, whereas a configuration only contains a list of components and the connections between their interface methods. Figure 2 shows the source code for the *TimerC* configuration, which contains a module named *TimerM* that implements the provided interface methods.

Actors are the major building blocks of a galsC program and are written in the galsC programming language. The interface of an actor consists of a set of input and/or output ports and a set of *parameters*. Parameters are global variables that can be both read and written. An actor contains a list of components and connections. A connection can connect a component interface method to one of the following endpoints: (1) another component interface method, (2) a port, (3) a parameter, or (4) some combination of these. We discuss connections in detail in Sections II-C and III. An actor may also contain an *actorControl* section which exports the *StdControl* interface of any of its components to the application level for system initialization (e.g. for initializing hardware components). Figure 3 shows the source code for the *SenseActor* actor.

A galsC program is created by writing a galsC *application* file that contains zero or more parameters and a list of actors and connections. A connection can connect application parameters (global names) with actor parameters (local names). A connection can also connect actor

```
configuration TimerC {
    provides interface Timer[uint8_t id];
    provides interface StdControl;
} implementation {
    components TimerM, ClockC, ...;
    TimerM.Clock -> ClockC; ...
    StdControl = TimerM.StdControl;
    Timer = TimerM.Timer;
}
```

```
module TimerM {
    provides interface Timer[uint8_t id];
    provides interface StdControl;
    uses interface Clock;
    ...
} implementation {
    // Interface implementation...
}
```

Fig. 2. Source code for the TimerC and TimerM components.

```
actor SenseActor {
    port {
        in trigger;
        out output;
    } parameter {
        uint16_t count;
    } implementation {
        components SenseToInt, Photo;

        SenseToInt.ADC -> Photo;
        SenseToInt.ADCControl -> Photo;

        trigger -> SenseToInt.trigger;

        (SenseToInt.IntOutput.output,
            count) -> output;

        actorControl {
            SenseToInt.StdControl;
        }
    }
}
```

```
application SenseTag {
    parameter {
        uint16_t count = 0;
    } implementation {
        actor TimerActor, SenseActor,
            ...;

        count = TimerActor.count;
        count = SenseActor.count;

        TimerActor.trigger =[64]=>
            SenseActor.trigger;
        SenseActor.output => ...;

        appstart {
            SenseActor.trigger();
        }
    }
}
```

Fig. 3. Source code for the SenseActor actor and the SenseTag application.

output ports with actor input ports, with an optional declaration of the port queue size (defaults to size 1). Figure 3 shows the source code for the SenseTag application.

B. Language semantics

Generally speaking, galsC implements the TinyGALS programming model as described in [1], which we summarize here. We have also developed an improved way to handle TinyGUYS (parameters), which differs from the model described in [1].

1) Ports: Each input port of an actor has a FIFO queue. Communication between actors occurs asynchronously through these queues. When a component within an actor calls a method that is linked to an output port, the arguments of the call are converted into events called *tokens*. A copy of the token is placed in the event queue of each input port connected to the output port. Later, the scheduler removes the token from the queue and calls the method that is linked to the input port with the contents of the token as its arguments. Thus, the queue separates the flow of control between the actors; the call to the output port returns immediately, and the component within the actor can proceed. The scheduler processes tokens in the order in which they are generated. Tokens are dropped if the input port queue is full; the programmer is currently responsible for selecting the correct size.

2) Parameters: The TinyGALS programming model has the advantages that actors become decoupled through message passing and are easy to develop independently. However, each message passed will trigger the scheduler and activate a receiving actor, which may quickly become inefficient if there is global state that must be updated frequently. The TinyGUYS (Guarded Yet Synchronous) mechanism provides a way for actors to share global data safely. This is implemented as the `parameter` feature in the galsC programming language.

With the TinyGUYS mechanism, actors may read a parameter synchronously (without delay). However, writes to the parameter are asynchronous in the sense that all writes are buffered. The buffer is of size one, so that the last writer to the parameter wins. Parameters are updated by the scheduler only when it is safe (i.e., after an actor finishes executing and before the scheduler triggers the next actor).

Parameters have global names that are mapped to the local parameter names of each actor. In the new TinyGUYS mechanism, a component interface method or an actor port can write to a parameter by calling a connected function with a single argument. Parameter values can be read by passing them as arguments to component interface methods or actor ports. In *SenseActor* in Figure 3, the *count* parameter is passed as the last argument to the *output* port.

This design does not require parameter names to appear inside the component name space. One can develop components in their own scope, independent of the connected parameters. We no longer require components to use special methods to access global variables, which greatly improves the reusability of components.

C. Connection model within actors

A connection $x \rightarrow y$ inside an actor consists of a source x and a target y.[1] We use regular expressions to describe possible entities of x and y:

$$source = (l)^* \ (p \mid f) \ (l)^* \quad (1)$$

$$target = l \mid p \mid f \quad (2)$$

where l is the local name of a parameter, p is an actor port name, and f is a component interface function. A *trigger* is a port or function that appears as the source of a connection. A port is triggered when the scheduler invokes it with the first token in its queue. A function is triggered when it is called by another function.

A connection $x \rightarrow y$ is valid if the number of arguments and the types of the arguments of the source match those of the target when the arguments on each side of the arrow are concatenated separately, similar to the notion of record types [9]. Additionally, a source port must be an input port and a target port must be an output port, and a source function must be a required method and a target function must be a provided method. The return type of a trigger must also match that of the target.

For example, suppose f_1 is a required method with exactly two arguments. $(f_1, l_1) \rightarrow p_1$ is valid if p_1 is an output port that has exactly three arguments whose types match those of the left hand side (i.e., the types of the first two arguments of p_1 must match those of f_1, and the type of the last argument of p_1 must match that of l_1) and if the return type of f_1 matches that of p_1.

Using our regular expression model, we have the following valid types of connections, where l in (t, l) is an abbreviation for any number of parameters appearing before or after the trigger t:

No parameters	Param GET	Param PUT
$p_1 \rightarrow p_2$	$(p_1, l) \rightarrow p_2$	$p \rightarrow l$
$p_1 \rightarrow f_1$	$(f_1, l) \rightarrow p_1$	$f \rightarrow l$
$f_1 \rightarrow p_1$	$(p_1, l) \rightarrow f_1$	Param GET/PUT
$f_1 \rightarrow f_2$	$(f_1, l) \rightarrow f_2$	$(p, l_1) \rightarrow l_2$
		$(f, l_1) \rightarrow l_2$

For connections with no parameters, the trigger will (a) trigger the connected function or (b) pass a token to the connected output port. In a parameter GET connection, the parameter value(s) will be appended to the trigger's argument list and passed to the connected function or port. In a parameter PUT connection, the trigger will write its argument to the parameter. In a parameter GET/PUT connection, the trigger will cause the source parameter to be read and its value stored in the target parameter. Note that for the number of arguments to match, the trigger in a parameter PUT connection must have only one argument, and the trigger in a parameter GET/PUT connection must have no arguments.

What are the semantics of multiple connections (i.e., fanout from a function)? For example, what is the order of computation when you have $f_1 \rightarrow l_1$ and $f_1 \rightarrow f_2$? Or when you have $f_1 \rightarrow l_1$ and $f_1 \rightarrow p$? In galsC, the write to the parameter occurs first, before any additional computation or transfer of control. The buffered parameter value may then get overwritten in the later computation. This policy gives us a consistent view of ordering in the system.

[1]This model also applies to connections at the application level. However, the discussed port directions must be reversed: a source port must be an output port and a target port must be an input port. Global parameter names should be used instead of local parameter names.

D. Type inference and type checking

The galsC compiler performs high level type inferencing on the connection graph of an application. There are two parts to the type inference system: connections with ports, and connections with parameters but no ports.[2]

1) Ports: In galsC, ports are untyped. The actual types of ports are inferred from the connection graph of a galsC program. In figure 4, an actor A contains a component which has a call to function f with type signature τ_1. The input port of actor B is the target of the concatenation of the output port of A with a parameter with type τ_3. The known types $(\tau_1, \tau_3, \tau_5, \tau_8)$ are shown in bold.

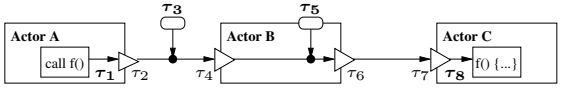

Fig. 4. Type checking example.

We write a type equation for each connection in the system:

$$\boldsymbol{\tau_1} = \tau_2 \quad (3)$$

$$\tau_2 \times \boldsymbol{\tau_3} = \tau_4 \quad (4)$$

$$\tau_4 \times \boldsymbol{\tau_5} = \tau_6 \quad (5)$$

$$\tau_6 = \tau_7 \quad (6)$$

$$\tau_7 = \boldsymbol{\tau_8} \quad (7)$$

We can then solve the set of equations to determine the types of the ports. A valid system has a unique solution to the set of equations. The galsC compiler derives types for all ports in the system by matching the return type and the argument types of all connected upstream and downstream functions. The galsC compiler detects a type error when the set of equations conflicts with itself or is unsolvable.

2) Parameters: The type system for parameter connections without ports is straightforward, since there are only two types of connections: (1) connections between a global name and a local name, and (2) connections between a function and a local name. Since the types of all these sources and targets are known, the type checker merely verifies that all the types in a connection match.

E. Code generation

The highly structured architecture of galsC programs enables us to automatically generate the communication and scheduling code, allowing software developers to avoid writing error-prone concurrency control code. We have extended the nesC 1.1.1 toolset [10]. The resulting galsC toolset can compile both nesC and galsC programs and its output can be cross-compiled for any platform used with TinyOS [11], including the Berkeley motes [12].

The TinyGALS code generation tools described in [1], were compatible with TinyOS 0.6.1 and were implemented in perl and generated stylized C (using C preprocessor macros). The galsC compiler, on the other hand, takes advantage of a real compiler backend. It uses traditional compiler techniques, including type checking, dead code elimination, and function inlining. We have modified the data-race detection feature of nesC, since the decoupling of execution through ports eliminates some possible sources of race conditions.

[2]Connections containing only functions are checked with the nesC type checker.

galsC provides an improved programming model in exchange for a minimal application-dependent increase in code size for scheduling and communication between actors [13]. For a simple photosensor application, the initialization and scheduling code is 662 bytes compared to 564 bytes for the original nesC code. The *get()* and *put()* functions for a port of type `uint8_t` use 208 bytes. The *get()* and *put()* functions for a parameter of type `uint16_t` use 30 bytes. The scheduler event queue size is equal to the sum of the user-allocated size for each port connection (depends on the size of the data type).

III. Concurrency Issues

Concurrency management is a significant concern in event-driven systems. Poorly implemented systems may suffer from deadlock (i.e. where no tasks can proceed due to blocking on a shared resource), livelock (i.e. where the system falls into deadloop and responds to no further interrupts), and race conditions (i.e. where shared variables are accessed by multiple threads at the same time).

In this paper, we only consider concurrency issues on single processor platforms. In galsC, all memory is statically allocated; there is no dynamic memory allocation. A galsC program runs in a single thread of execution (single stack), which may be interrupted by the hardware. An actor A may begin execution when: (1) the scheduler activates A in response to an event in its input port, or (2) an interrupt service component within A is triggered by an external interrupt. The execution triggered by interrupts is called the *interrupt context*, and the execution activated by the scheduler is called the *scheduled context*. Interrupt handlers preempt scheduled executions, which is the only source of concurrent execution in galsC. Our system-level concurrency model allows us to manage the concurrency issues discussed earlier.

A. Cross-actor concurrency

Since all scheduled executions of actors are in the scheduled context and controlled sequentially by the scheduler, the only possibility for cross-actor concurrent execution is when one actor is in the scheduled context, and one or more other actors are in an interrupt context.

There are two mechanisms for actors to communicate in galsC: event queues (ports) and guarded global variables. Blocking on shared resources (e.g., a blocking read) is not part of the semantics across actors, which gives us:

Theorem 1: Deadlock is not possible across actors.

In event-driven systems, since there are critical system operations, such as enqueuing and dequeuing events, which are atomic, it is possible for a scheduler to retain control and disable interrupts indefinitely. In Figure 5, the *Loop* actor is first triggered by an internal interrupt, which produces an event (token) at the output port. The event loops back to the input port where it is inserted into the event queue. Interestingly, there is a direct link between the input port and the output port inside the actor. Can this self-loop prevent further interrupts from entering the system?

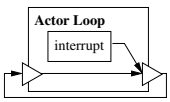

Fig. 5. A self-loop actor triggered by an interrupt.

Once the event is enqueued, the scheduler (1) dequeues the event, with interrupts disabled and (2) calls the function connected to the inside of the input port, in this case the *put()* function of the output port. Within the *put()* function, the code that inserts the event back into the event queue is also atomic. So, without a careful implementation of the scheduler, there is a risk of livelock. However, in the galsC scheduler, interrupts are enabled between dequeuing the event and enqueuing the event, so future interrupts will not be blocked, which gives us:

Theorem 2: Livelock is not possible across actors.

Race conditions are another major concurrency concern. Since there are shared data between actors, an actor may be in the middle of writing the data when another actor tries to read it. Two actors may also try to write to a shared variable at the same time.

There are two forms of shared data across actors: tokens and parameters. Tokens are stored in event queues, and access to them is atomic and controlled by the scheduler. Parameters, as discussed in the previous section, are always guarded, whose value updates are again controlled by the scheduler (where the last value written wins). Thus,

Theorem 3: Race conditions are not possible across actors.

As a result of these claims, concurrency errors will not happen at the application level across actors. So, programmers can focus on concurrency issues within each actor, which is a problem with a much smaller scope.

B. Component-level concurrency

Concurrent execution may occur within actors if multiple threads of control enter the same component through its input ports or interrupt handlers. Problems may occur if they try to change the component state at the same time.

A piece of code is *reentrant* if multiple simultaneous, interleaved, or nested invocations do not interfere with each other. We assume that interrupts handlers are not reentrant, and that interrupts are masked while servicing them (interleaved invocations of the same interrupt are disabled). However, other (different) interrupts may occur while servicing an interrupt. Hardware interrupts are the only sources of preemption.

A component C may begin execution if: (1) the hardware that C encapsulates interrupts (C is a "*source component*"), (2) an event arrives on the actor input port linked to one of the interface methods of C ("*triggered component*"), or (3) another component calls one of the interface methods of C ("*called component*"). Once activated, a component executes to completion. That is, the interrupt service routine or method finishes.

Reentrancy problems may arise if a component is both a source component and a triggered component. An event on a linked actor input port may trigger the execution of a component method. While the method runs, an interrupt may arrive, leading to possible race conditions if the interrupt modifies internal variables of the same component.

Cycles within actors (between components) are not allowed, otherwise reentrant components are required.[3] Therefore, any valid configurations of components within an actor can be modeled as a directed acyclic graph (DAG). A *source DAG* is formed by starting with a source component and following all forward links between it and other components in the actor (Figure 6(a)). A *triggered DAG* is similar to a source DAG but starts with a triggered component instead (Figure 6(b)). Race conditions and reentrancy problems may occur if source DAGs and triggered components are connected within an actor. In Figure 6(c), the source DAG (C_1, C_3) is connected

[3]Recursion within components is allowed. However, the recursion must be bounded for the system to be live.

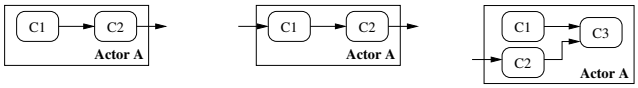

Fig. 6. a) Source DAG b) Triggered DAG c) Source DAG & triggered DAG

to the triggered DAG (C_2, C_3). Race conditions and reentrancy problems may occur if C_3 is running in a scheduled context and an interrupt causes C_1 to preempt C_3. We can relax the restriction on cycles between components and only disallow cycles in method call chains between components by first separating the methods within a component into separate source and triggered components.

If all interrupts are masked during interrupt handling (interrupts are disabled), then we need not place any additional restrictions on source DAGs. However, if interrupts are not masked (interrupts are enabled), then a source DAG must not be connected to any other source DAG within the same actor. Triggered DAGs can be connected to other triggered DAGs, since with a single thread of execution, it is not possible for a triggered component to preempt a component in any other triggered DAG.

Some configurations of connections between component methods and actor ports may lead to nondeterministic component firing order. Let us first assume that both actor input ports and actor output ports are totally ordered (we use the order specified in the `port` section of the galsC actor file), but that components are not ordered. Then actor input ports may either be associated with one (provided) method of a single component C or with one or more actor output ports. Likewise, outgoing component methods (required) may be associated with either one (provided) method of a single component C or with one or more actor output ports.[4] Provided component methods may be associated with any number or combination of required component methods and actor input ports, but they may not be associated with actor output ports. Likewise, actor output ports may be associated with any number or combination of required component methods and actor output ports. However, if we assume that neither actor input ports nor actor output ports are ordered, then actor input ports and outgoing component methods may only be associated with either a single method or with a single output port.

The components within a single galsC actor must satisfy the following conditions to be well-formed and avoid concurrency problems:

- Source components may not also be triggered components nor called components.
- Cycles among components within an actor are not allowed, but loops around actors are allowed.
- Component source DAGs and triggered DAGs must be disconnected.
- Component source DAGs must not be connected to other source DAGs, but triggered DAGs may be connected to other triggered DAGs. Assumes that an interrupt whose handler is running is masked, but other interrupts are not masked.
- Outgoing component methods may be associated with 1 method of another component, or with $\geq$1 output ports.

[4]In the existing TinyOS constructs, one caller (outgoing component method) can have multiple callees. The interpretation is that when the caller calls, all the callees will be called in a possibly non-deterministic order. A combination of the callees' return values will be returned to the caller. Although multiple callees are not part of the TinyGALS semantics, it is supported by our software tools for TinyOS compatibility.

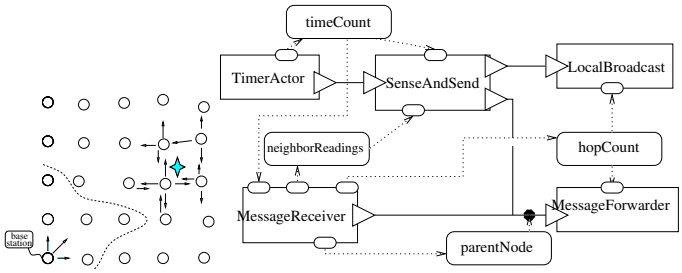

Fig. 7. a) Sensor array for object detection and reporting. b) Top-level, per-node view of the object detection application.

- Input ports may be associated with 1 method of a single component, or with $\geq$1 output ports.

C. Determinacy

Notice that lacking of concurrency errors does not mean galsC programs are deterministic. The *system state* of a galsC program consists of (1) the internal state of all components, (2) the contents of the global event queue[5] and (3) the values of all global parameters. The question of *determinacy* is that given a unique initial state of a galsC program and a set of known interrupts (in terms of both interrupt time and value), will the program have a unique state trajectory independent of the execution/CPU speed? Note that single thread sequential programs, where all inputs are read into the system, are determinate. Concurrent models, such as Kahn process networks, which sacrifices real-time properties, can also be determinate [14]. However, for event-driven systems, determinacy may be sacrificed for reactiveness.

In general, a galsC program is non-determinate. The source of non-determinacy is the preemptive handling of interrupts. Suppose that while an actor is being iterated, it is interrupted by another actor. If both of these actors produce events at their output ports, the order of events in the global event queue may not be consistent when the system is executed at different speeds. If both of these actors write to a global variable (i.e. parameter), then without exact timing information, we cannot predict the final value of the global variable at the end of the iteration.

IV. Example

To illustrate the effectiveness of the galsC language, let us consider a classical sensor network application that detects and monitors point-source targets. A set of sensor nodes (called motes) are deployed in a 2-D field. To simplify the discussion, we assume that the motes are deployed on a perturbed grid, as shown in Figure 7(a). The goal of the sensor network is to detect moving objects modeled as point signal sources, and to report the detection to a central base station, located at the lower-left corner of the field. Please note that the goal here is to illustrate the language, rather than to develop sophisticated algorithms to solve the problem optimally.

We assume that the motes know their locations on the grid and the grid size. The application primarily consists of two tasks: exchanging local sensor readings to determine the "leader" responsible for reporting a detection, and multi-hop forwarding of the report messages to the base station. For simplicity, the leader election is achieved by having every mote periodically broadcast a packet containing the

[5]The global event queue is defined as the ordered sequence of tokens in the event queues of all actor ports.

location of the mote and its sensor reading. These packets also serve as beacons to establish a multi-hop routing structure.

The multi-hop routing is implemented as a routing tree rooted at the base station. Assume that no mote has the global topology of the network; a mote finds out its parent in the tree by eavesdropping on other messages. These messages include sensor reading broadcasts and forwarded report messages. Every message contains the hop count of the sender, which indicates the level of the sender in the routing tree. For example, the mote directly connected to the base station has hop count 0. Whenever it broadcasts a message, everyone who can overhear the message will note that it is probably one hop away from the base station. As illustrated by the dashed line in Figure 7(a), the reachable nodes of a wireless broadcast may have a complicated shape. To compensate for the unreliable and sometimes asymmetric wireless communication links, a mote maintains a list of senders it has heard in the past T seconds and chooses the most reliable one (measured by, for example, a trade-off between low hop count and message repeatability) as its parent node. It then calculates its own hop count from its parent's hop count.

A high-level view of the implementation of the object detection application in galsC is shown in Figure 7(b). All motes run the identical code modular to their locations. The execution of a mote is driven by two event sources – clock interrupts and received messages. Similar to the example in Figure 1, the *TimerActor* handles clock interrupts and updates the latest timer count in a parameter named *timeCount*. Every half second, *TimerActor* emits a token that triggers the *SenseAndSend* actor.

The *MessageReceiver* actor receives messages from the radio and chooses an action based on the message type:

- If the message is a local broadcast, it updates the *neighborReadings* table. Note that since only the latest neighbor sensor reading matters, the overriding semantics of TinyGUYS variables is a natural fit.
- Also for each broadcast message, it updates an internal routing table by looking at the repetition frequency of the sender node. Note that it requires the *timeCount* value to determine the rate of the messages heard. Whenever there is a change of the desired parent node, and thus this node's hop count, it updates the *parentNode* and *hopCount* parameters.
- If the message is a forwarding message, it sends the content of the message to the downstream *MessageForwarder* actor.

The *SenseAndSend* actor activates the ADC to get a sensor reading. Once the sensor reading is available, it queues a local broadcast of the sensor reading. It also compares its own reading with the latest values from its neighbors.[6] If this mote has the highest sensor reading (i.e. it is closest to the signal source), *SenseAndSend* generates a report message and queues it with the *MessageForwarder* actor.

Both the *LocalBroadcast* actor and the *MessageForwarder* actor send out packets with this mote's *hopCount* so that other motes can use it to build the multi-hop routing tree. The *MessageForward* actor also takes the *parentNode* ID as part of the input token, merged with the requests from *SenseAndSend* and *MessageReceiver*.

V. Conclusion and Future Work

This paper describes galsC, a language for event-driven embedded systems that allows developers to use high-level constructs such as *ports* and *parameters* to create thread-safe, multitasking programs. We have created a type system for checking connections across synchronous and asynchronous communication boundaries. The galsC compiler automatically generates communication and scheduling code for programs specified in the galsC language, which allows developers to avoid writing error-prone task synchronization code. Our compiler backend also allows traditional type checking, dead code elimination, and function inlining, as well as checking for possible race conditions. The language and compiler are implemented for the Berkeley motes and extend TinyOS/nesC by providing a higher level programming abstraction than the TinyOS primitives.

There are several directions for further research. Currently, developers must use trial and error to find appropriate port queue sizes to avoid token loss when the buffers are full. We could instrument the generated galsC code and scheduler to determine the rate at which tokens are generated and automatically determine the best queue sizes for the application.

Ptolemy II (ptII) [15] is a Java-based system for modeling and simulating heterogeneous concurrent systems. We could generate galsC/nesC code from within ptII, allowing programmers to model and simulate within a graphical interface and move seamlessly to running programs in the field. The message passing and parameter semantics of galsC can support many other models of computation in ptII, such as synchronous dataflow, discrete event, Kahn process networks, and Giotto. galsC code generated from ptII could contain a replacement scheduler to implement these other models of computation, while taking advantage of its system-level type system [16].

The current galsC language and compiler only address programs running on a single node. Communication across multiple nodes is a natural extension of asynchronous communication in galsC between actors on a single node. Developers could then write programs for entire sensor networks, rather than programs for individual nodes, which can be difficult and unintuitive without specialized knowledge about the specific node and its interactions with other nodes.

[6] Here, the neighbors are defined as the motes directly above, below, left, and right of this mote in the grid.

References

[1] E. Cheong, J. Liebman, J. Liu, and F. Zhao, "TinyGALS: A programming model for event-driven embedded systems," in *SAC'03*, pp. 698–704.

[2] D. Gay *et al.*, "The nesC language: A holistic approach to networked embedded systems," in *PLDI 2003*.

[3] N. Halbwachs, *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[4] D. B. Stewart *et al.*, "Design of dynamically reconfigurable real-time software using port-based objects," *IEEE Trans. on Software Engineering*, pp. 759–776, December 1997.

[5] W. A. Najjar, E. A. Lee, and G. R. Gao, "Advances in the dataflow computational model," *Parallel Computing*, January 1999.

[6] F. Balarin *et al.*, "Synthesis of software programs for embedded control applications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 834–849, June 1999.

[7] J. Bhasker, *A SystemC Primer*, 2nd ed. Star Galaxy Pub.

[8] W. LaRue *et al.*, "Functional and performance modeling of concurrency in vcc," in *Concurrency and Hardware Design : Advances in Petri Nets*. LNCS 2549, Springer-Verlag Heidelberg, 2002, pp. 191 – 227.

[9] M. Wand, "Type inference for record concatenation and multiple inheritance," in *4th Annual Symposium on Logic in Computer Science*, 1989.

[10] "nesC compiler," http://sourceforge.net/projects/nescc/.

[11] "TinyOS: a component-based OS for the networked sensor regime," http://www.tinyos.net/.

[12] Crossbow Technology, Inc., http://www.xbow.com/.

[13] E. Cheong and J. Liu, "galsC: A language for event-driven embedded systems," University of California, Berkeley, Memorandum UCB/ERL M04/7, April 2004.

[14] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of the IFIP Congress 74*, 1974, pp. 471–475.

[15] "The Ptolemy project," http://ptolemy.eecs.berkeley.edu.

[16] E. A. Lee and Y. Xiong, "System-level types for component-based design," in *EMSOFT 2001*, October, pp. 237–253.