galsC: A Language for Event-Driven Embedded Systems

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

Jie Liu
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304
Email: jieliu@parc.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. In galsC programs, all inter-actor communication, actor triggering mechanisms, and access to guarded global variables are automatically generated by the compiler. 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

Embedded software designers face issues such as maintaining consistent state across multiple tasks, handling interrupts, avoiding deadlock, managing concurrent threads, 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.

Published as: Memorandum No. UCB/ERL M04/7, 20 April 2004, Electronics Research Laboratory, College of Engineering, University of California, Berkeley 94720.

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. For related work and more information on the TinyGALS programming model, please see [1].

In this paper, we introduce *galsC*, a language that implements the TinyGALS programming model. Improved from our previous work [1] based on TinyOS 0.6.1, our new language takes advantage of the nesC [2] specification for TinyOS 1.x [3]. Having a real compiler backend allows us to further develop a type system across synchronous and asynchronous communication mechanisms and analyze race conditions. The galsC compiler and toolsuite is built on the nesC 1.1.1 compiler and toolsuite [4] for the wireless sensor network nodes known as the Berkeley motes [5]. nesC/TinyOS components provide an interface abstraction that is consistent with synchronous communication via method calls. Unlike our model, however, concurrent tasks in TinyOS are not exposed as part of the component interface. Lack of explicit management of concurrency forces component developers to manage concurrency by themselves (locking and unlocking semaphores), which makes TinyOS applications difficult to develop. Our system is compatible with TinyOS 1.x components. galsC programs use the TinyGALS scheduler, which runs the TinyOS 1.x scheduler at the lowest priority for backwards compatibility.

To some extent, galsC is closer to system-level hardware/software codesign languages, such as SystemC [6] and VCC [7], than embedded software languages such as nesC. It 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 the galsC syntax and semantics, connection model, type checking system, and code generation system. Section III discusses concurrency issues in galsC programs. Section IV describes a sample application implemented in galsC. Section V discusses the effects of interrupts on determinacy in galsC programs and describes directions for future work.

II. The galsC Language

In this section we describe the galsC syntax and semantics. We use a simple sensing application (shown in Figure 1) to illustrate these concepts. In this example, a hardware clock at a high rate triggers the system to update a counter of time ticks. Downsampled clock signals trigger the system to read the light intensity level from a photoresistor at a lower rate. Reading the sensor may take time. The resulting sensor value is then tagged with the latest value of the counter and sent for further downstream processing.

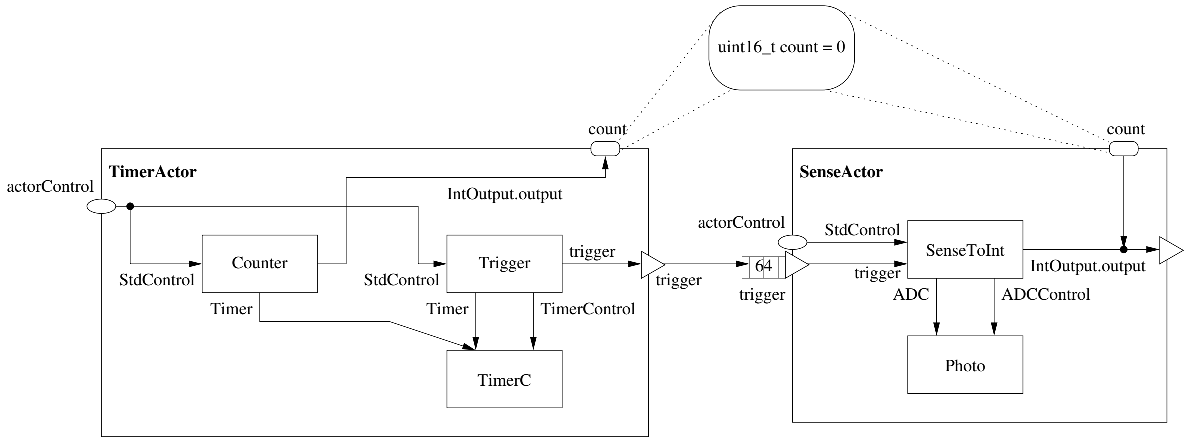


Fig. 1. The SenseTag application.

A. Language constructs

There are three basic constructs in galsC: components, actors, and applications.

1) Components: Components are the most basic elements of a galsC program and are written in the nesC programming language. Components provide and/or require[1] *interfaces*, which are collections of methods. A component that provides an interface contains an implementation of the interface method(s), whereas a component that uses an interface expects another component to implement the interface. A component is either a *module* or a *configuration*. A module contains actual system implementation code, whereas a configuration only contains a list of components and the connections between the component interface methods.

Figure 2 shows the source code for a configuration named `TimerC`. It contains a module named `TimerM` that actually implements the provided interface methods.

[1]In nesC, a required interface uses the `uses` keyword.

```
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.

2) Actors: 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 with one of the following endpoints: (1) another component interface method (like the connections in a component implementation), (2) a port, (3) a parameter, or (4) some combination of these. Restrictions on connections are discussed 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, starting, and stopping hardware components).

Figure 3 shows the source code for `TimerActor`, which contains the `TimerC` component, whose source code was shown in Figure 2. `TimerActor` has an output port named `trigger`, which is connected to a component interface method. The `count` parameter is written to by a component interface method, `Counter.IntOutput.output`. `TimerActor` exports the `Counter.StdControl` and `Trigger.StdControl` interfaces for system initialization.

Figure 3 also shows the source code for `SenseActor`. The output port `output` is connected to the concatenation of the component interface method `SenseToInt.IntOutput.output` and the value read from the `count` parameter. Figure 1 shows a graphical representation of the source code.

3) Application: A galsC program is created by writing a galsC `application` file. An application 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 output ports with actor input ports, with an optional declaration of the port queue size.

Figure 4 shows the source code for the SenseTag application, which contains `TimerActor`, `SenseActor`, and some downstream actors. The application contains a parameter named `count`, which is initialized to zero and connected to the corresponding parameters of `TimerActor` and `SensorActor`. The output port `trigger` of `TimerActor` is connected to the corresponding input port of `SenseActor`, with a queue size of 64. The `appstart` section declares that an initial token is placed in the input port of `SenseActor`.

```
actor TimerActor {
    port {
        out trigger;
    } parameter {
        uint16_t count;
    } implementation {
        components Counter, TimerC, Trigger;

        Counter.Timer ->
            TimerC.Timer[unique("Timer")];
        Counter.IntOutput.output -> count;

        Trigger.Timer ->
            TimerC.Timer[unique("Timer")];
        Trigger.TimerControl -> TimerC;
        Trigger.trigger -> trigger;

        actorControl {
            Counter.StdControl;
            Trigger.StdControl;
        }
    }
}
```

```
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;
        }
    }
}
```

Fig. 3. Source code for TimerActor and SenseActor.

B. Language semantics

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]. galsC programs are designed to run with the TinyGALS scheduler.

1) Ports: In the TinyGALS programming model, each input port 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 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.

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

```
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. 4. Source code for the SenseTag galsC application.

writer to the parameter wins. Parameters are updated by the scheduler only when it is safe (i.e., after a scheduled 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. In `TimerActor` in Figure 3, the `Counter.IntOuput.output` method has a single argument which is written to the `count` parameter whenever the method is called. Parameter values can be read by passing them as arguments to component interface methods or actor ports. In `SenseActor`, the `count` parameter is passed as the last argument to the `output` port.

This new design does not require parameter names to appear inside of the component name space. Components can be developed in their own scope, independently of which parameters are connected. By doing so, we no longer require components to use special methods to access global variables. This greatly improves the reusability of components.

C. Connection model within actors

A connection $x \to y$ inside an actor consists of a source x and a target y.[2] We use regular expressions to describe possible entities of x and y:

[2]This model also applies to connections at the application level (inside a galsC application file). 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. Note that functions do not appear at the application level.

$$source \;=\; (l)^{*}\ (p \mid f)\ (l)^{*} \tag{1}$$

$$target \;=\; l \mid p \mid f \tag{2}$$

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 [8]. 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 right hand side (i.e., the types of the first two arguments of p_1 must match those of the arguments 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:

- Without parameters
  - $p_1 \rightarrow p_2$ [When the input port p_1 is triggered, transfer data directly from p_1 to the output port p_2.]
  - $p_1 \rightarrow f_1$ [When the input port p_1 is triggered, trigger a function f_1.]
  - $f_1 \rightarrow p_1$ [When the function f_1 is triggered, create a token from the arguments of the function f_1 and send it to the output port p_1.]
  - $f_1 \rightarrow f_2$ [When the function f_1 is triggered, trigger another function f_2.]
- With parameters
  - Parameter GET
    - $(p_1, l) \rightarrow p_2$ [When the input port p_1 is triggered, concatenate the current value of the parameter(s) l with the arguments of p_1 and send the resulting token directly to the output port p_2.]
    - $(f_1, l) \rightarrow p_1$ [When the function f_1 is triggered, concatenate the current value of the parameter(s) l with the arguments of f_1 and send the resulting token to the output port p_2.]
    - $(p_1, l) \rightarrow f_1$ [When the input port p_1 is triggered, concatenate the current value of the parameter(s) l with the arguments of p_1 and trigger a function f_1 with the corresponding arguments.]
    - $(f_1, l) \rightarrow f_2$ [When the function f_1 is triggered, concatenate the current value of the parameter(s) l with the arguments of f_1 and trigger another function f_2 with the corresponding arguments.]
  - Parameter PUT

    - $p \to l$ [When the input port p is triggered, write its argument to a parameter l.]
    - $f \to l$ [When the function f is triggered, write its argument to a parameter l.]
- Parameter GET/PUT
    - $(p, l_1) \to l_2$ [When the input port p is triggered, read the current value of the source parameter l_1 and write it to the target parameter l_2.]
    - $(f, l_1) \to l_2$ [When the function f is triggered, read the current value of the source parameter l_1 and write it to the target parameter l_2.]

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 connections 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 $f_1 \to l_1$ and $f_1 \to f_2$? Or when you have $f_1 \to l_1$ and $f_1 \to p$? 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.

D. *Type checking*

The galsC compiler performs high level type checking on the connection graph of an application. There are two parts to the type checking system: connections with ports, and connections with parameters but no ports.[3]

1) Ports: In galsC, ports are untyped. The actual types of ports are inferred from the connection graph of a galsC program, which we explain using the example in Figure 5. Actor A contains a component which makes 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 output port of B is the target of the concatenation of the input port of B and a parameter with type τ_5. The output port of B is directly connected to the input port of actor C. The input port of C is a trigger for a function with type signature τ_8. The known types $(\tau_1, \tau_3, \tau_5, \tau_8)$ are shown in bold.

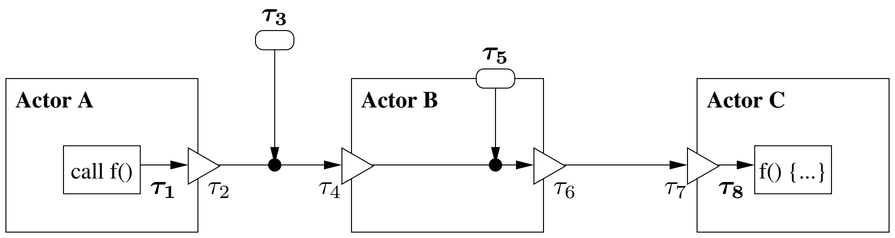


Fig. 5. Type checking example.

We write a type equation for each connection in the system. Thus we have the following set of equations:

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

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

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

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

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

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

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 checking system for parameter connections without ports is quite 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 of these sources and targets are known, the type checker merely verifies that all of the types in a connection match each other.

E. Code generation

The highly structured architecture of the TinyGALS/TinyGUYS programming model allows us to automatically generate the communication and scheduling code for galsC programs. This allows software developers to avoid writing error-prone concurrency control code. We have extended the nesC 1.1.1 compiler and toolset [4]. The resulting galsC compiler and toolset can compile both nesC and galsC programs and its output can be cross-compiled for any platform used with TinyOS [3], including the Berkeley motes [5].

In [1], we described the code generation tools for TinyGALS, which was compatible with TinyOS 0.6.1. These tools were implemented in perl and generated stylized C (using C preprocessor macros). However, the new galsC compiler is highly improved, since we can take advantage of having a real compiler backend. The galsC compiler uses traditional compiler techniques, including type checking, dead code elimination, and function inlining. We also inherit the data-race detection feature of nesC. However, this must be modified for galsC, since the decoupling of execution through ports eliminates some possible sources of race conditions. We also use the connection model described in Section II-C to check connections and infer and check types in the system graph of ports, parameters, and functions.

The galsC compiler generates functions and variables (shown in Tables I and II), which are equivalent to those described in [1]. The sizes (memory usage) of these generated functions and variables are comparable to those in [1]. The new TinyGALS and TinyOS scheduler sizes are comparable as well.

TABLE I

GENERATED CODE FOR PORTS IN GALSC

Function or variable name	Per port[4]	Function	Description
`GALSC_sched_init()`		X	Initialize scheduler data structures.
`GALSC_sched_start()`		X	Put initial tokens into input port queues.
`GALSC_eventqueue[]`			Event queue for the TinyGALS scheduler.
actor$*port*$`put()`	X	X	Put token into input port queue.
actor$*port*$`get()`	X	X	Get token out of input port queue.
actor$*port*$`arg`i`[]`	X		Queue for the ith argument of the input port.[5]
actor$*port*$`head`	X		Points to the beginning of the input port queue.[5]
actor$*port*$`count`	X		Keeps track of how many tokens are in the input port queue.

TABLE II

GENERATED CODE FOR PARAMETERS IN GALSC

Function or variable name	Per parameter[6]	Function	Description
`GALSC_params`			Structure that contains all of the parameters (TinyGUYS) in the program.
`GALSC_params_buffer`			Copy of `GALSC_params`.
parameter$`put()`	X	X	Write to parameter buffer.
parameter$`get()`	X	X	Read from parameter.

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. There are two cases in which an actor A may begin execution: (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.

[4]"Per port" indicates that this function or variable is generated for each input port. Otherwise, there is only one instance of the function or variable for the entire galsC program.

[5]This variable is not generated if the port has no arguments (i.e., the token contains no data).

[6]"Per parameter" indicates that this function or variable is generated for each parameter. Otherwise, there is only one instance of the function or variable for the entire galsC program.

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:

Claim 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. Let us consider the configuration shown in Figure 6.

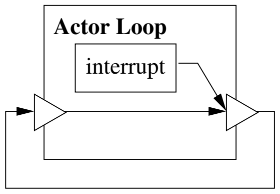


Fig. 6. A self-loop actor triggered by an interrupt.

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?

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:

Claim 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. Thus,

Claim 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 also occur within an actor, especially for actors that have both input ports and interrupt handlers. Since nesC/TinyOS components are like objects, which encapsulate private variables and methods that may change the value of these variables, the main concurrency concern is that multiple threads of control may enter the same component.

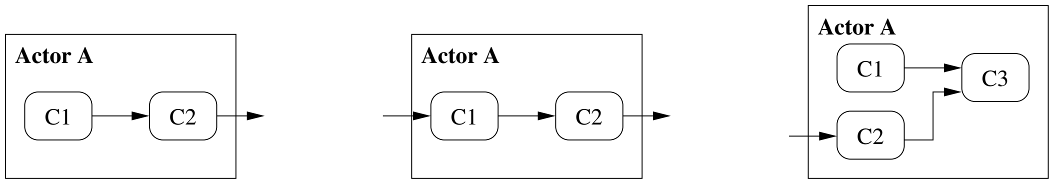
A piece of code is *reentrant* if multiple simultaneous, interleaved, or nested invocations do not interfere with each other. In this section, we assume that interrupts handlers are not reentrant, and that interrupts are masked while servicing them (interleaved invocations of the same interrupt are disabled). However, other (different) interrupts may occur while servicing an interrupt. There are no other sources of preemption other than hardware interrupts. When using components that contain interrupt handlers in which interrupts are enabled, we must take special care in placing constraints on what constitutes a valid configuration of components within an actor in order to avoid unexpected reentrancy, which may lead to race conditions and other nondeterminacy issues. Methods that do not access component state will not suffer from race conditions, but may suffer from reentrancy problems. To simplify our discussion, we assume all methods may potentially access component state.

There are three cases in which a component C may begin execution: (1) an interrupt from the hardware that C encapsulates, (2) an event arrives on the actor input port linked to one of the interface methods of C, or (3) another component calls one of the interface methods of C. In the first case, the component is a *source component* and when activated by a hardware interrupt, the corresponding interrupt service routine is run. Source components do not connect to any actor input ports. In the second case, the component is a *triggered component*, and the event triggers the execution of a provided method. Both source components and triggered components may call other components via required methods, which results in the third case, where the component is a *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. Therefore, to improve the ease of analyzability of the system and eliminate the need to make components reentrant, source components must not also be triggered components, and vice versa. The same argument also applies to source components and called components. Therefore, it is necessary that source components only have outputs (required methods) and no inputs (provided methods).

Cycles within actors (between components) are not allowed, otherwise reentrant components are required.[7] Therefore, any valid configurations of components within an actor can be modeled as a directed acyclic graph

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



(a) A source DAG is activated by a hardware interrupt.

(b) A triggered DAG is activated by the arrival of an event at the actor input port.

(c) When a source DAG are connected to a triggered DAG, race condition and reentrancy problems may occur.

Fig. 7. DAGs within actors.

(DAG). Figure 7(a) shows a *source DAG* in an actor A, which is formed by starting with a source component C in A and following all forward links between C and other components in A. Figure 7(b) shows a *triggered DAG* in an actor A, which is formed by starting with a triggered component C in A and following all forward links between C and other components in A. In general, if source DAGs and triggered DAGs are connected within an actor, race conditions and reentrancy problems may occur. In Figure 7(c), the source DAG (C_1, C_3) is connected 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.

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. Recall that once triggered, the components in a triggered DAG will execute to completion. We must also place restrictions on what connections are allowed between component methods and actor ports, since some configurations may lead to nondeterministic component firing order.

Let us first assume that both actor input ports and actor output ports are totally ordered (we assign the order to be the same as the order specified in the `port` section of the galsC actor file). However, we assume that components are not ordered. As discussed earlier, the configuration of components inside of an actor must not contain cycles and must follow the rules above regarding source and triggered DAGs. 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.[8] 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.

If 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.

In summary, in order to avoid reentrancy problems, race conditions, and other concurrency bugs, we have developed conditions for well-formed galsC actors that are free of concurrency problems:

- Source components must only have outputs; they may not have inputs. In other words, 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.
- Within an actor, component source DAGs and triggered DAGs must be disconnected.
- Within an actor, component source DAGs must not be connected to other source DAGs, but triggered DAGs may be connected to other triggered DAGs. We assume that an interrupt whose handler is running is masked, but other interrupts are not masked.
- Within an actor, outgoing component methods may be associated with either one method of another component, or with one or more actor output ports.
- Within an actor, actor input ports may be associated with either one method of a single component, or with one or more actor output ports.

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 8. 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 then developing sophisticated algorithms to solve the problem optimally.

We assume that the motes know their own locations and the grid size at the deployment, and that their clocks are reasonably synchronized. 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 location of the mote and its sensor reading. These packets also serve as beacons to establish a multi-hop routing

[8]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.

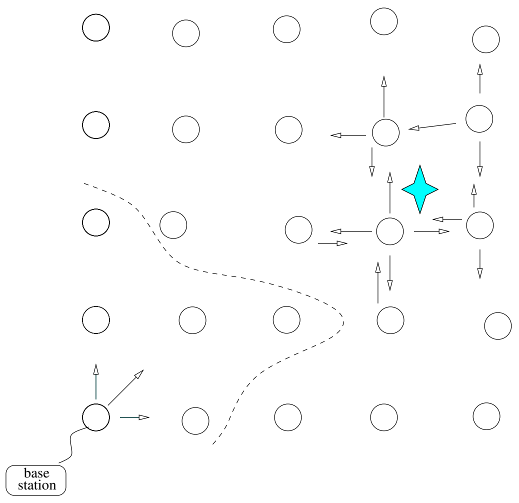


Fig. 8. An array of sensors for object detection and reporting.

structure.

The multi-hop routing is implemented as a routing tree rooted at the mote connected to 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. Each 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 8, 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. Implementation in galsC

The high-level view of the implementation of the object detection application is shown in Figure 9. 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.

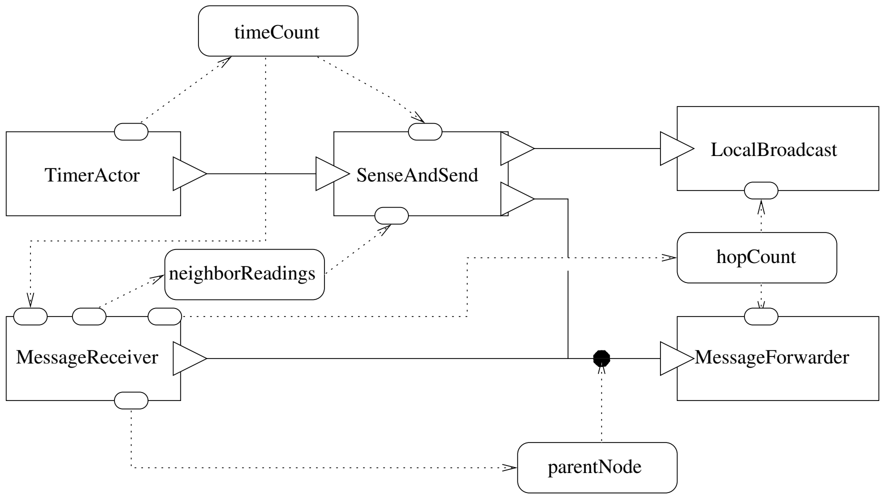


Fig. 9. The top-level view of the object detection application.

- Also for every broadcast message, it updates an internal routing table by looking at the repetition 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.[9] 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. Discussion

A. Determinacy

The *system state* of a galsC program consists of (1) the internal state of all components, (2) the contents of the global event queue[10] and (3) the values of all global parameters. The problem 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), whether

[9]Here, the neighbors are defined as the motes directly above, below, left, and right of this mote, for a total of four motes.

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

the program will 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, can also be determinate [9]. However, for event-driven systems, determinacy may be sacrificed for reactiveness.

To analyze the determinacy property of galsC programs, let us define a system state as *quiescent* if there are no events in the global event queue, otherwise it is *active*. An application is partitioned into actors. An *iteration* of an actor is the execution of a subset of the components inside of the actor in response to either an interrupt or an event at an input port. Thus, a typical system execution starts from a quiescent state, iterates through several actors, and ends at a quiescent state.

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, then without exact timing information, we cannot predict the final value of the global variable at the end of the iteration.

A partial solution for reducing non-determinacy in the system is to delay producing outputs from the actor being iterated until the end of its iteration. This approach is taken by models of computation such as timed multitasking [10] and Giotto [11]. If we know the order of interrupts, then we can predict the state of the system after a single actor iteration even if it is interrupted one or more times.

A galsC program is determinate in a restricted case, where we have pure reactive execution. That is, interrupts occur only at quiescent states. This may require that the processing speed be quick enough to process all triggered execution before the next interrupt occurs. An extreme version of this case is the “synchronous” assumption in synchronous/reactive models, where the processing speed is infinitely fast, and it takes zero time to react to external events [12].

B. Future directions

In galsC applications, the software developer specifies the port queue sizes, otherwise the compiler generates a buffer of default size one. Currently, the developer must use a trial and error process to determine appropriate queue sizes to avoid token loss when the buffers are full. However, we could instrument the generated galsC code and scheduler to determine the frequency at which tokens are generated at actor output ports. With a system-in-the-loop configuration, we could use the frequency data to determine the best queue sizes automatically for a given application.

galsC programs are examples of hierarchical heterogeneous systems [13]. The procedure call model of computation within actors forms the lower level of the hierarchy. The message passing model of computation between actors forms the higher level of the hierarchy. Ptolemy II [14] is a system written in Java for modeling and simulating heterogeneous concurrent systems. An interesting direction to pursue would be to add a modeling domain to Ptolemy II to allow users to create galsC programs within the Ptolemy II graphical interface. Adding a galsC domain to Ptolemy II would also allow us to take advantage of its system-level type system [15]. We could automatically generate galsC and nesC code from within Ptolemy II, allowing programmers to move seamlessly from modeling and simulating to running programs in the field. The message passing (tokens) and parameter semantics can support many other models of computation implemented in Ptolemy II, such as SDF (synchronous dataflow), DE (discrete event), PN (Kahn process networks), and Giotto. galsC code generated from Ptolemy II could contain a replacement scheduler to implement these other models of computation.

Although galsC is currently designed for single node systems, the asynchronous communication between actors is a natural fit for inter-node communication. We plan to explore ways to extend the TinyGALS programming model to work across multiple nodes. Developers can 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.

VI. Conclusion

This paper describes galsC, a language for event-driven embedded systems that implements the TinyGALS programming model. This model allows software designers 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 us to do traditional type checking, dead code elimination, and function inlining, as well as checking for possible race conditions. The language and compiler use TinyOS/nesC and are implemented for the Berkeley motes.

Acknowledgements

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.

References

[1] E. Cheong, J. Liebman, J. Liu, and F. Zhao, "TinyGALS: A programming model for event-driven embedded systems," in *Proceedings of the Eighteenth Annual ACM Symposium on Applied Computing*, March 2003, pp. 698–704.

[2] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," in *Proceedings of Programming Language Design and Implementation (PLDI) 2003*, June 2003, http://webs.cs.berkeley.edu/tos/papers/nesc.pdf.

[3] "TinyOS: a component-based OS for the networked sensor regime," http://webs.cs.berkeley.edu/tos/.

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

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

[6] J. Bhasker, *A SystemC Primer, Second Edition*. Star Galaxy Publishing, 2004.

[7] W. LaRue, S. Solden, and B. Bhattacharya, "Functional and performance modeling of concurrency in vcc," in *Concurrency and Hardware Design : Advances in Petri Nets*, J. Cortadella, A. Yakovlev, and G. Rozenberg, Eds. LNCS 2549, Springer-Verlag Heidelberg, 2002, pp. 191 – 227.

[8] M. Wand, "Type inference for record concatenation and multiple inheritance," in *Fourth Annual Symposium on Logic in Computer Science*. Asilomar Conference Center, Pacific Grove, CA: IEEE Computer Society Press, 1989, pp. 92–97. [Online]. Available: citeseer.ist.psu.edu/wand89type.html

[9] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of the IFIP Congress 74*, International Federation for Information Processing. Paris, France: North-Holland Publishing Company, 1974, pp. 471–475.

[10] J. Liu and E. A. Lee, "Timed multitasking for real-time embedded software," *IEEE Control Systems Magazine*, pp. 65–75, February 2003.

[11] T. Henzinger, B. Horowitz, and C. Kirsch, "Embedded control systems development with Giotto," in *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES'01)*, June 2001.

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

[13] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity—the Ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, January 2003.

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

[15] E. A. Lee and Y. Xiong, "System-level types for component-based design," in *First International Workshop on Embedded Software (EMSOFT 2001)*, ser. Lecture Notes in Computer Science, T. A. Henzinger and C. M. Kirsch, Eds., vol. 2211. Springer-Verlag, October 2001, pp. 237–253.