A Component-based Model and Language for Wireless Sensor Network Applications

Alan Dearle, Dharini Balasubramaniam, Jonathan Lewis, Ron Morrison
School of Computer Science, University of St Andrews
{al, dharini, jonl, ron}@cs.st-andrews.ac.uk

Abstract

Wireless sensor networks are often used by experts in many different fields to gather data pertinent to their work. Although their expertise may not include software engineering, these users are expected to produce low-level software for a concurrent, real-time and resource-constrained computing environment. In this paper, we introduce a component-based model for wireless sensor network applications and a language, Insense, for supporting the model. An application is modelled as a composition of interacting components and the application model is preserved in the Insense implementation where active components communicate via typed channels. The primary design criteria for Insense include: to abstract over low-level concerns for ease of programming; to permit worst-case space and time usage of programs to be determinable; to support the fractal composition of components whilst eliminating implicit dependencies between them; and, to facilitate the construction of low footprint programs suitable for resource-constrained devices. This paper presents an overview of the component model and Insense, and demonstrates how they meet the above criteria.

1. Introduction

A wireless sensor network (WSN) [1] consists of a number of devices or nodes, which communicate and cooperate to achieve a common goal. These devices are typically small and resource constrained. The goal of a WSN application is customarily the generation, transmission and processing of data pertinent to the application domain.

The users (and programmers) of WSNs are often not software engineers but domain experts in other fields, such as ecology or geology, who use WSNs to obtain data relevant to their work. Despite this disparity in expertise, they are required to operate in a concurrent, real-time, resource constrained computing environment that is potentially complex and hard to program correctly.

Figure 1 shows a UML deployment diagram representing a simple application that periodically captures temperature readings from a temperature sensor, averages the captured readings and sends the averages to a temperature logger. The application comprises three component instances joined by two channels.

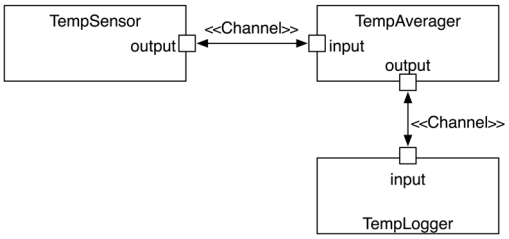

Figure 1: Temperature Deployment Diagram

In most systems the process of creating an implementation involves a loss of application model integrity with many concepts and concerns impacting the design. As concerns, such as memory management and concurrency control, are addressed, the implementation diverges from the original application model.

In this position paper, we propose a simple yet powerful component-based model and a new language, Insense [2], which realises the model, in an attempt to simplify the development of WSN applications. In our model the basic building block is the component. A component:

- has no implicit dependencies on other components;
- is stateful but there is no sharing of any state between components;
- has a single thread of control and a behaviour that specifies the component's semantics; and
- may be composed with other components using Fractal composition [3] – a component can instantiate other components (which cannot be seen from outside the instantiating component).

As we demonstrate in this paper, the model and language contain appropriate abstractions which

obviate the need for application programmers to deal with low-level issues such as memory management and concurrency control. Our model uses channels to provide a single typed communication mechanism, which abstracts over both communication and synchronisation. By contrast, many examples of WSN programming found in the literature involve writing programs to poll devices, set time-outs and perform low-level synchronisation between processes and devices.

The remainder of the paper describes how the component model is realised in Insense, and briefly outlines the major implementation challenges.

2. The Insense language

The Insense language is a realisation of the component-based model described above. As such, Insense applications are constructed as compositions of active components, which communicate via channels. Our key aim is to reduce the complexity of WSN applications by abstracting over programming complexities such as synchronisation, memory management and event-driven programming.

Insense integrates a number of novel features including: permitting worst-case space and time usage of programs to be determinable; supporting the fractal composition of components using the dependency injection pattern [4], whilst eliminating implicit dependencies between components; and, facilitating the construction of low footprint programs suitable for execution on devices with constrained resources.

In Insense, a component is a stateful object and contains updatable locations. These locations may only be accessed by the locus of control that is implicitly defined by the component's behaviour, which is akin to a thread that never leaves the component. Thus each component is a unit of concurrent computation. The elimination of sharing ensures that accidental race conditions and other synchronisation errors cannot occur.

A component can create instances of other components, and thus components can be arranged in a Fractal pattern. Due to the encapsulation rules, components created inside a component can only be referenced by their creator. To both preserve architectural decisions in the implementation and ensure that unintended sharing of variables cannot occur, a component cannot reference any external data objects or locations. To enforce this, the only sharable data structure in the language is the channel through which all inter-component communication takes place.

In Insense, channels are typed and directional. All values in the language may be passed along a channel including components and other channels. Furthermore, arbitrary values may be encoded in an infinite union type called *any* [5] and passed across suitably typed channels. Channel communication is synchronous. Send operations block until the message is received and receive calls block when there is no input to be read.

The type of a component is represented by an *interface* that describes the names and types of the channels presented by the component for communication. The *TempAverager* component from Figure 1 presents two channels: an *in* channel of type *integer* and an *out* channel also of type *integer*. A component receives values of the appropriate type from an *in* channel, and sends values on an *out* channel.

The type of the interface presented by *TempAverager* in Figure 1 may be written as:

```
type averagerIF is interface( in integer input;
                              out integer output )
```

Figure 2: A Component Interface Type

Note that an interface declaration merely describes the type of a component; it does not define the component or its behaviour. A component definition has four parts: a specification of the interface it presents, definitions of component variables, at least one constructor, and exactly one behaviour.

```
component TempAverager presents
                          averagerIF {
    size = 4
    store = new integer[size] of 0
    index = 0
    constructor() {
    }
    behaviour {
        receive next from input
        store[index] := next
        index := index + 1
        if( index >= size ) {
            send average(store) on output
            index := 0
        }    }   }
```

Figure 3: The *TempAverager* Component

Figure 3 shows the definition of the *TempAverager* component from Figure 1. A component definition is introduced by the keyword *component* which is followed by the component's name. The name of the component is always followed by the keyword *presents*, which must be followed by a (comma-

separated) list of the interfaces it presents. *TempAverager* presents a single interface of type *averagerIF* as shown in Figure 2. The names of all channels in the presented interfaces are implicitly declared. In this example, the names *input* and *output* are brought into scope and are of types "*in integer*" and "*out integer"* respectively.

Component instances are created by calling one of the component's constructors. Like constructors in Java, they may perform arbitrary computation but are normally used to perform constructor dependency injection by initialising component variables.

The keyword *behaviour* introduces the behaviour of the component, which repeats until the component is stopped either by itself or by another component using the keyword *stop*. In Figure 3 the behaviour first reads integers from the *input* channel using the **receive from** construct. It stores the received values in an array, and when four readings have been received it writes the average on the *output* channel. Note that the component contains the definitions of component-local variables called *size*, *store* and *index*. In Insense the types of declarations are inferred by the compiler and the "=" symbol is used to initialise the variable with a value. Variables are local to the component in which they are declared and may not be accessed outwith this component. The scope of component variables is from the point they are declared until the end of the component.

The behaviour of an instance begins execution as soon as the instance is created. Communication between components may be initiated by connecting the output of one component instance to the input of another. To illustrate instantiation and connection, we define instances of components from Figure 1 and connect them, as shown in Figure 4.

```
sensor = new TempSensor ()
averager = new TempAverager ()
logger = new TempLogger ()
connect sensor.output to averager.input
connect averager.output to logger.input
```

Figure 4: Creating and Connecting Components

This would complete the assembly of three components shown in Figure 1.

Since there is no shared store in Insense programs, channels are the only mechanism by which concurrent loci of execution (component behaviours) may synchronise and interact. While the semantics of individual send and receive operations are deterministic, non-determinism occurs in the language since multiple output channels may be connected to a single input and vice versa. Thus the order in which messages are communicated is determined by scheduling.

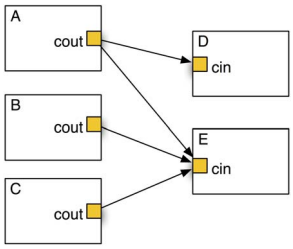

Figure 5: The Communication Topology

Figure 5 shows an example of a connection topology where an output channel *cout* in component *A* is connected to input channels *cin* in both components *D* and *E*. Similarly the input channel *cin* in component *E* is connected to output channels *cout* in components *A*, *B* and *C*. The ability to connect multiple input and output channels enables the specification of complex communication topologies.

```
select {
    receive x from chan1 when p > 7 : p := x
    receive y from chan2 when p < 7 : p := y
    receive z from chan3: p := z
    default: p := 1}
```

Figure 6: Non-deterministic Select

Insense supports the *select* construct, a powerful guarded non-deterministic selection over multiple channels, an example of which is shown in Figure 6. The semantics of the construct is perhaps best explained with this example. The select clause non-deterministically reads from one of the three channels *chan1*, *chan2* and *chan3*.

A select arm is eligible for execution only if an input is available on the channel specified in the arm and the (optional) *when* clause associated with the arm evaluates to true. If none of the arms are eligible for execution, the (optional) default is executed. If no default is specified, and no arms are eligible for execution, the construct blocks until an input is available on at least one of the arms.

Insense supports the following additional control constructs: *if-then-else* and *switch* statements for choice, a bounded *for* loop for iteration and a *try-except* clause for exception handling.

For WSN programming, Insense provides bindings between the language and the hardware platform on which it is executing. Predefined components are used

to model hardware with their interface representing inputs to and outputs from devices. For example, a TMote Sky might be modelled as a component with channels representing the temperature, humidity and light sensors.

Finally, in a resource-constrained environment that typifies WSN applications, it is important to be able to reason about the space and time usage of programs. In Insense, the space requirements of components, and consequently those of Insense programs, are statically determinable. Thus it is possible to reason about there being enough space resources to run a program prior to its execution or prior to changes being made when programs are evolved. Similarly, it is possible to reason about the time bounds of components so that (soft) real-time schedules may be adhered to.

3. Implementation

The first version of the Insense compiler is written in Java and generates C source code suitable for compilation by gcc. The code generated for an Insense component defines a factory function for a C object which follows the Microsoft COM component pattern described by Box [6]. Each Insense component logically extends an object called *IComponent_data* as shown in Figure 7.

Each component contains a pointer to the functions that operate upon it and a pointer to the process implementing the behaviour of that component. The functions defined in the function table include the component's behaviour.

```
typedef struct IComponent_data {
    struct IComponent_funcs *funcs;
    struct process *process_pntr;
    struct IComponent_data *blocked;
    // Type specific data below here
} *ICompPNTR, ICompStruct;
```

Figure 7: IComponent Implementation

For each programmer-defined component a C type which logically extends *IComponent_data* is defined. Whenever an Insense component is created during execution, an instance of one of these structs is created by the appropriate constructor. It contains fields for all the variables defined in the component. Since Insense functions may not allocate space, this struct statically defines the total space budget for the component and no additional stack or heap space needs to be allocated. The runtime representation of an instance of the *TempAverager* component shown in Figure 3 is shown in Figure 8. As in COM, the function table (VTBL) is shared by all instances of the component.

Note that space is statically allocated for (pointers to) the channels used, the component's global variables and the behaviour's local variables, obviating the need for a stack. Note also that the creation of a component is an inexpensive operation akin to object creation in a language such as Java.

As can be observed in Figure 8, the worst case space usage for each component may be statically calculated by adding the space requirements of each of the component's local and global variables and adding the space required for the process structure and each of the component's channels.

The worst-case time usage may be calculated by inspection of the component's behaviour. Since no recursive functions or while loops are provided by the language, this is again a simple calculation.

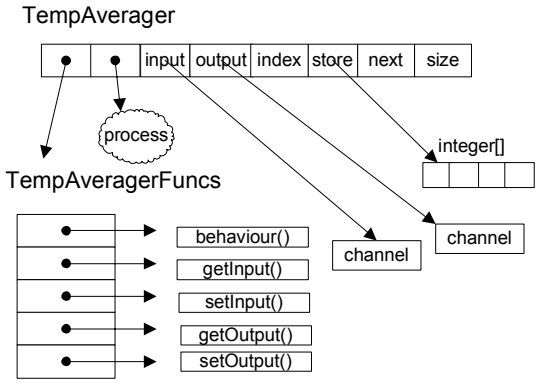

Figure 8: TempAverager Run-time Structure

We now turn our attention to the implementation of channels before returning to components and concurrency.

Each Insense channel is implemented by two *half channels* with each half channel being owned by the Insense component in which it was declared. The two-part implementation of channels is required in order to allow the possibility of a number of output channels being connected to a single input channel and vice versa, as shown in Figure 5.

Each half channel contains five fields:

- a *buffer* for storing one item of the channel type;
- *ready* and *nd_received* flags which indicate whether their owner is ready to communicate and whether data was received during a *select* respectively;
- a list of pointers, called *connections*, to the half channels to which the half channel is connected;
- two binary semaphores: one called *mutex* which serialises access to the half channel and;

- another called *blocked* upon which senders and receivers may block.

When a channel is declared in the language, a corresponding half channel is created in the implementation. Whenever a connection is made, a pointer is added to each of the lists pointing to the corresponding half channel.

Insense supports four operations on channels: *connection*, *disconnection*, *send* and *receive* (with non-deterministic selection being a special case of *receive*). When a connection is established, each half channel is locked in turn using the *mutex*. Next a reference to the corresponding half channel is added to the *connections* list and the *mutex* released. Disconnection traverses the *connections* list and dissolves the bi-directional link between the half channels.

```
send( data : int, half_channel cout ) {
    wait( cout.mutex )
    set( cout.ready )       // signal sender is ready
    signal( cout.mutex )
    foreach( halfchan match in cout.connections )
    {
      wait( match.mutex ) // start with receiver
      if( match.ready ) {   // a receiver is ready
        match.buffer = data // copy to receiver
        unset( match.ready ) // rcvr matched
        set(match.nd_received) // used by selectors
        signal( match.blocked ) // let rcvr run
        signal( match.mutex ) // finished with rcvr
         wait( cout.mutex ) // got match so
        unset( cout.ready )          // clear ready
        signal( cout.mutex )
        return
      }
      signal( match.mutex ) // finished with rcvr
     }
    cout.buffer = data     // save in sender buffer
    wait( cout.blocked )} // block sender
}
```

Figure 9: The Send Algorithm

The most interesting operations are the *send* and the *receive*, shown in Figures 9 and 10 respectively. They are almost symmetric. Both operations attempt to find a waiting component in the list of connections with the receiver looking for a waiting sender and vice-versa. If no such match is found the sender or receiver block on the *blocked* semaphore. We informally demonstrate the correctness of these operations. If the sender is forced to wait (due to no receiver being ready), the datum being sent is stored in the sender's buffer. It is copied into the receiver's buffer when a receiver attempts to find a match. Conversely if a receiver is forced to block, the sender copies the datum into the receiver's buffer when it finds a match. In both cases the datum ends up in the receiver's buffer. Both operations set the owner's *ready* flag whose update is protected by a *mutex*. This flag is either cleared during the operation, if a match is found, or by the corresponding operation.

If an operation is forced to block, it does so on the owner's *blocked* semaphore. In this case it can be seen by inspection that no semaphores are left set. When a thread does wait on *blocked* it is always after the *ready* flag has been set, after the half channels in the *connections* list have been checked and, in the case of the sender, when the data is in the send buffer. When a match is made, the *ready* flag of the waiting process is cleared and the data is copied to the appropriate buffer.

```
receive( half_channel cin ) {
    wait( cin.mutex )
    set( cin.ready )         // signal receiver ready
    signal( cin.mutex )
    foreach( halfchan match in cin.connections )
    {
      wait( match.mutex ) ) // start with sender
      if( match.ready ) {   // a sender is ready
        cin.buffer = match.buffer // copy from sndr
        unset( match.ready )        // sndr matched

        signal( match.blocked ) // let sender run
        signal( match.mutex )  // finished with sndr
        wait( cin.mutex ) // got match so
        unset( cin.ready ) // clear ready
        signal( cin.mutex )
        return
      }
      signal( match.mutex ) // finished with sender
    }

    wait( cin.blocked ) }  // block receiver
}
```

Figure 10: The Receive Algorithm

The non-deterministic select operation is interoperable with the send operation shown in Figure 9, and has three stages of execution. The first stage establishes select arms that are eligible for execution and constructs a list of channels and integer pairs representing these arms. If the list is not empty a random list entry is chosen and used to execute the corresponding receive operation and right hand side of the arm. If no arms were eligible for execution in the first stage, the second stage executes the default arm if one has been specified. If there is no default arm the

last stage sets a *ready* flag that is temporarily shared by each channel specified in the select arms for which the guard conditions in the when clause are satisfied. The operation then un-sets all *nd_received* flags and blocks until data is sent on one of the channels by waiting on a *blocked* semaphore that is also temporarily shared by the channels. The *nd_received* flag is used by the select code to determine from which channel a datum was received and to execute the appropriate code. The last stage completes by restoring the original *blocked* semaphores and *ready* flags in each half channel.

4. Operating system interaction

We have implemented Insense using the Contiki operating system [7] with each Insense behaviour mapped onto a Contiki lightweight process.

Contiki protothreads [8] (and the lightweight process mechanism built above them) do not support automatic (stack) variables in C. However, this does not impact the Insense implementation since space is allocated for component variables when component instances are constructed.

The unit of scheduling is an Insense component and when a component is scheduled the component state needs to be recovered. We have achieved this by encoding this address in the process name which obviates the need to modify the Contiki scheduler. Whenever a process is scheduled the component state is inexpensively extracted from the process name and assigned to a variable called *this*. All component variables are referenced using this pointer.

5. Conclusions

We have introduced a component-based model and a language called Insense, for developing wireless sensor network applications. WSN applications are often concurrent, real-time, and resource constrained. The primary aim of this work is to provide a simple programming model for such systems, which abstracts over the complexity of the execution environment and associated low-level programming concerns.

This approach preserves architectural design decisions throughout the implementation that are often lost in the complexities required to deal with low-level concerns. This yields applications that are independent of specific operating systems and hardware platforms.

The language is also designed to allow the calculation of worst-case space and time usage of programs. The requirements of an Insense application can be statically determined, reducing the possibility of execution time errors.

Components, which are the main building blocks of Insense applications, interact with one another by communicating via typed channels. This communication model abstracts over concurrency and synchronisation concerns, simplifying application development. Fractal composition of components is facilitated by using the dependency injection pattern whilst eliminating dependencies between components.

We have demonstrated an efficient implementation of the channel concept by using half-channels to deal with rich channel connection topologies.

At this stage we offer the Insense language and implementation as a proof of concept. A working implementation of the language exists, and is being used as the basis for further development and evaluation. The next stage in this work is to design evaluation metrics in terms of ease of programming, flexibility, applicability, portability and efficiency.

Acknowledgments

This work is supported by the EPSRC grant entitled DIAS-MC (Design, Implementation and Adaptation of Sensor Networks through Multi-dimensional Co-design) EP/C014782/1.

References

[1] C.S Raghavendra, K.Sivalingam, T. Znati, (eds): Wireless Sensor Networks. Kluwer Academic Press (2004).

[2] A. Dearle, "Insense Manual," University of St Andrews Report, 2007. URL: http://dias.dcs.st-and.ac.uk/inSense/manual.pdf

[3] E. Bruneton, T. Coupaye, and J. B. Stefani, "The Fractal Component Model," ObjectWeb, 2004.

[4] M. Fowler, "Inversion of Control Containers and the Dependency Injection Pattern," 2004.

[5] R. Morrison, A. Brown, R. Carrick, R. Connor, A. Dearle, and M. Atkinson, "Polymorphism, Persistence and Software Reuse in a Strongly Typed Object Oriented Environment," Software Engineering Journal, pp. 199-204, 1987.

[6] D. Box, Essential COM. Addison-Wesley, 1997.

[7] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," presented at Proceedings of The First IEEE Workshop on Embedded Networked Sensors, Tampa, Florida, 2004.

[8] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems," presented at Proceedings ACM SenSys 2006, Boulder, Colorado, 2006.