Integration of Event-Driven Embedded Operating Systems Into OMNet++ – A Case Study with Reflex

Sören Höckner
LS Verteilte Systeme/Betriebssysteme
BTU Cottbus
Cottbus Germany
shoeckne@informatik.tu-cottbus.de

Andreas Lagemann
LS Verteilte Systeme/Betriebssysteme
BTU Cottbus
Cottbus Germany
ae@informatik.tu-cottbus.de

Jörg Nolte
LS Verteilte Systeme/Betriebssysteme
BTU Cottbus
Cottbus Germany
jon@informatik.tu-cottbus.de

ABSTRACT

Developing wireless sensor network (WSN) applications is a challenging task. Simulations are a key component in the development process, since they offer simple means of testing and evaluating the applications without the need of time consuming and tedious deployment. But simulations alone are not sufficient to evaluate such applications. Only experiments on real hardware can ultimately verify the correctness of a given algorithm and its implementation. To take the most benefit from a WSN simulator it must be able to simulate a sensor network, where all nodes run the same implementation of the algorithm that will later be deployed. We show how to integrate event-driven operating systems into the *OMNeT++* discrete event simulator. At the example of Reflex we show how an integration can be easily achieved with minor effort. Additionally we discuss an alternative approach which promises better scalability but comes at the cost of less flexibility at the application layer and requires more deeply intrusions into the operating system. We argue that the integration is feasible and that it yields a simulation tool, which can perform similar to other tools like TOSSIM or COOJA but benefits notably from the flexibility of *OMNeT++* and its cornucopia of readily available models provided by the community.

Categories and Subject Descriptors

C.3 [**Special-purpose and Application-based Systems**]: Realtime and embedded systems; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Experimentation

Keywords

Simulation, OMNeT++, Sensor networks

OMNeT++ 2009, Rome, Italy.

1. INTRODUCTION

Wireless Sensor Networks (WSN) typically consist of dozens to hundreds of tiny micro controller based computers (usually called motes or nodes) equipped with a set of problem specific sensors and actuators. Additionally every node has a transceiver allowing for wireless communication. Typical sensor network applications are *sense and send* and *monitoring*. In *sense and send* applications every single node simply samples its sensor readings and periodically sends the data to a well known sink, where the data is processed. In a *monitoring* application individual nodes evaluate their sensor readings and react by issuing warning messages, when the values are not within predefined bounds. Both applications demonstrate the event-driven nature of WSN. That means that the behavior of sensor network applications is dependent on the set of events that they are exposed to and on the possible set of interactions induced by each particular ordering of events.

This implies that programs for WSN cannot be sufficiently tested let alone evaluated on a single node. The authors of [13] postulate that the right simulation tool for WSN must allow to study the whole application, including the operating system (OS) and the network stack. Furthermore these tools, according to [13], need to handle large numbers of nodes while providing sufficient detail to capture the subtle effects caused by unpredictable interference and noise. According to the authors, many researchers have concluded that fulfilling all these requirements simultaneously is intractable. They have therefore instead applied very abstract simulations for instance by adapting *ns-2* ([16], [21] and [7]). While these are valuable simulation approaches they do not study the actual implementation of the WSN application, which actually hampers if not inhibits deployment, because you have to implement the algorithm twice: once for the simulator and once for the sensor node. This is not only very tedious, additionally it may lead to correct algorithms implemented wrongly. Guaranteeing consistency between two implementations is tremendously hard. Therefore it would be most advisable to have some means of using the same implementation for the simulation as well as for the target platform with at most some minor adaptations in a central location.

While there are many ways to achieve this (refer to Section 2), we chose to integrate our lightweight, event based operating system Reflex [20] into *OMNeT++* . This was a relatively easy task. The reason for that lies in the design

of both REFLEX and *OMNeT++* .

This paper presents the key properties of the design of *OMNeT++* and REFLEX used for the integration and illustrates how the integration was done. It is on the one hand intended to serve as a guideline to fulfilling similar tasks with similar operating systems resp. simulators. On the other hand it will identify enabling properties of the simulator and the OS which serve the goal of seamless integration of the two worlds.

The paper is structured as follows: The next section discusses related work in the field of simulating wireless networks. The following two sections give a short overview of REFLEX and *OMNeT++*. Then the integration is described in detail and possible alternatives are discussed. At last we conclude with possible future work.

2. RELATED WORK

The choice of an adequate simulator is a crucial decision for researchers in WSN, since the quality of the simulation directly correlates to the quality of the results to be published. A recent trend is that publications not relating to experimental results are rarely accepted at conferences with a WSN related topic. It would seem that more and more researchers will agree that simulation alone is not sufficient to evaluate solutions for WSN. Nevertheless simulation is an essential step in the development process. Since the deployment of WSN applications is very tedious and cumbersome, it is advisable to evaluate the application as accurately as possible before carrying out the deployment of the actual sensor network.

There have been many approaches to simulating WSN applications. Simulations can be roughly classified into three classes according to their level of abstraction. The first class operates on the network layer and abstracts from the operating system and the specific platform. One representative of this class is *ns-2* [5]. It is a general purpose network simulator which led to an immensely accelerated progress in the field of network protocol research by providing a common toolbox for studying a wide range of network protocols against various traffic models. *ns-2* has also been utilized for WSN simulations ([16], [21], [7]). The problem with these simulation approaches is that they remain too abstract and do not address issues with the actual implementation of the algorithms.

The second class operates at the operating system level. A common example of this class is TOSSIM [13], a discrete event simulator for sensor networks that use the TinyOS platform. With TOSSIM it is possible to compile TinyOS applications directly into its framework and then run thousands (according to [13]) of motes simultaneously executing the same application.

The third class enables simulation at the machine code level. A representative for this is Avrora [18]. Avrora provides cycle accurate simulation of the behavior of sensor network devices and communication. The authors claim that avrora was as scalable as TOSSIM while it was merely 50% slower. As the name suggests, it is restricted to AVR based microcontrollers. A similar tool exists for MSP430 microcontroller based nodes called MSPsim [10]. It is however not capable of simulating whole WSN as a standalone application but is merely intended to be used with a cross level simulator like COOJA [15].

COOJA has been designed to integrate all three categories into one modular extensible simulation framework by simulating code on all three levels simultaneously. Behavior at network level can be simply implemented in Java. The other two levels are based on the Contiki [9] operating system. It can be simulated on the OS level, where the Contiki application including all system components is compiled into a shared library. The simulator controls the application via the Java native interface. It can also be simulated on the machine code level by utilizing MSPsim.

OMNeT++ is a discrete event simulator like *ns-2* and *TOSSIM*. Unlike *TOSSIM* and *ns-2* it is very generic and modular, which allows it to model everything that fits the event driven approach. Therefore it offers a proper platform to integrate all kinds of simulation approaches. *OMNeT++* is already widely used as simulation tool for WSN. Several frameworks exist, which offer models useful for WSN simulation e.g. for wireless communication and node mobility ([2], [3], [1], [6], [4] among others). Due to the modular design of *OMNeT++* , components of the different modules can be mixed relatively easy, to fit specific needs. The integration of REFLEX into *OMNeT++* gives us access to a great number of modules supported by a large user community and offers the opportunity to profit from future developments made for *OMNeT++*, as they can be easily integrated with our current models.

3. REFLEX – THE REALTIME EVENT FLOW EXECUTIVE

REFLEX is an event driven operating system for deeply embedded systems and is particularly useful for embedded control applications as well as wireless sensor nodes. It is mostly written in C++ (aside from few parts of the platform specific code written in assembler) and therefore benefits from a sound object oriented design. It has a monolithic structure which means that both the program and the operating system are compiled into one single binary image. For space and performance reasons REFLEX is a single threaded system which uses a specific task model to represent independent activities. All tasks share the single system stack, therefore a task switch does not involve a stack switch. As a consequence all tasks must exhibit a run to completion semantic. This task model is well known and sufficient to model typical applications for embedded systems. Tiny OS [14] for instance is using this model as well. REFLEX, like Contiki [9], has an event driven architecture. The basic structural entity is the component. A component can contain activities (tasks) and interrupt handlers. They represent logical blocks which may for instance be composed to implement a complex state based control algorithm. The handlers and tasks provide active behavior and are therefore the entities managed by the scheduler. Handlers and tasks are connected by event channels. Events may contain data and are exchanged via these channels. Events are always buffered by the channels, including the possibly contained data. Whenever an event is assigned to a channel, the corresponding receiving task is added to the scheduler's ready list and eventually selected for activation. This strategy effectively guarantees that only such tasks are activated, which have valid input, i.e. their corresponding input channels have data available for processing.

Figure 1 illustrates the component based design of a REFLEX application, it is taken from [20]. Additionally to the

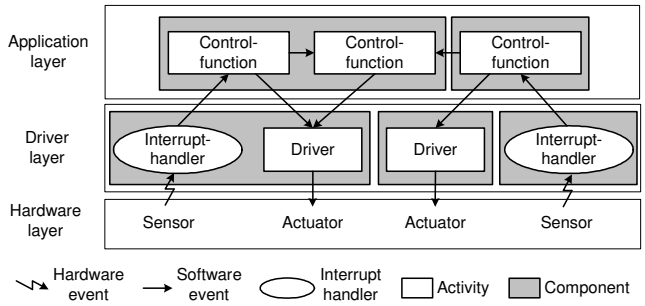

Figure 1: Reflex Event Flow Model

component structure each application can be divided into layers. The lowest layer is established by the hardware components themselves, which can be coarsely classified into sensors and actuators. The hardware for our example application consists of two sensors and two actuators. The driver layer contains components for the control of the devices present at the hardware layer. Tasks that are responsible for controlling devices are called drivers. Interrupt handlers are a special kind of task, which are not activated by the scheduler but directly from an interrupt, triggered by a sensor. They are the only entities that can be activated independently of the event flow. Therefore they should be kept as short as possible, merely take care of the pieces of work that cannot be postponed and defer the actual handling to an activity by issuing an appropriate event. It can be easily seen in figure 1 that the initial source of each event is always an interrupt. While the communication between nodes is completely asynchronous, the activities and handlers inside of components can share state information without explicit synchronization. This approach is similar to TinyGALS (Globally Asynchronous Locally Synchronous) [8]. It has several advantages, e.g. implicit synchronisation as has been shown in [11].

What distinguishes Reflex from other event-driven embedded operating systems, is its flexible scheduling framework [20], which gives the developer the choice between several scheduling strategies at compile time. It supports FCFS- (First Come First Serve), FP- (Fixed Priority), EDF- (Earliest Deadline First) and TT- (Time Triggered) scheduling. Some of these strategies support resp. require preemptive scheduling of tasks. When using preemptive scheduling schemes, some caution is indicated to preserve the implicit synchronization properties. For instance when using preemptive FP-scheduling (a non-preemptive version exists too), all tasks inside the same component which concurrently access their shared state must be assigned the *same* priority, otherwise synchronization has to be implemented explicitly.

Portability is a crucial property of operating systems for deeply embedded systems. Reflex provides support for easy porting by a strict modular organization of the source code. A large section of the core system code is written independently of specific hardware. All hardware dependent code is subdivided into a controller specific part and a platform specific part. A controller refers to a specific microcontroller architecture. A platform however refers to a specific device consisting of a microcontroller equipped with a set of devices (e.g. sensors, actuators, radio etc.). Controllers for which Reflex has been ported include MSP430, HCS12, MIPS32, Atmega128, Atmega8535, M16C, H8300 and linux for executing Reflex on i386 based systems (for testing or simulation). Platforms for which ports exist include CardS12, MICA2, OMNetPP, guest, ESB, MICA2DOT, SK-XC164CS, GLYNR8C, Mega128, RCX and TMoteSky.

4. OMNET++

OMNeT++ is a discrete event simulator written in C++. To execute simulations with *OMNeT++* a model needs to be defined, which consist of modules. There are two kinds of modules, simple modules and compound modules. The simple modules are basic building blocks. They are the active components of the model, programmed in C++ and represented by the `cSimpleModule` class, which serves as a base for the implementing class. Simple modules can be combined into compound modules and so forth. Both simple and compound modules are instances of module types. During model design, the developer declares module types, instances of which can be combined to more complex modules. At last, the system module is defined as an instance of any defined module type. Both kinds of module types can be transparently used as building blocks. This allows for restructuring a module instance by breaking it up into several module instances or synthesize several components into one module without affecting the interface of the building block. Modules communicate with messages which are typically sent via *gates* but may possibly be sent to their destination module directly (via method calls). Gates define the input and output interfaces of modules. A *channel* connects an input gate with an output gate. Connections are always contained within a single hierarchy level. The gates of a simple module may be either connected with corresponding gates of other simple modules within the same compound or with a gate of the enclosing compound module of the same type. Figure 2 (taken from [19]) demonstrates the structure of *OMNeT++* models. Boxes represent modules, arrows represent connections and small square boxes represent gates.

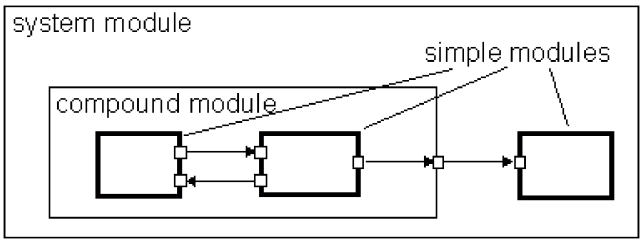

Figure 2: The model structure of *OMNeT++*

Events are represented as messages in *OMNeT++* . A separate event class is not provided. A message is an instance of the class `message` or any derived type. They are sent from one module to another; therefore the module where the event will occur is called the *message destination*. Parameters add more flexibility to the usage of modules and facilitate the reuseability of modules. They are mainly used to pass configuration data to simple modules and to help define model topology. A module can define an arbitrary number of parameters.

Since simple modules are the active elements of a model, a designer may redefine four methods defined in `cSimpleModule`. The `initialize()` method is called before the simulation start, in order to initialization of the module. If needed, an arbitrary number of initialization stages can be defined in order to solve dependency issues. The `handleMessage()` and the `activity()` methods are called during event processing. Each method represents a different approach to event processing. While `handleMessage()` is called each time a message is delivered at the corresponding module, `activity()` implements the process interaction approach. Each simple module can use exactly one of both alternatives. Modules implementing `handleMessage()` and such using `activity()` can be mixed freely. When using the coroutine based approach, the `receiveMessage()` method has to be used in order to receive messages. The call of `receiveMessage` is blocking, i.e. due to the cooperative nature of the coroutine approach this is the only point where the control is transfered to the simulation kernel and then possibly to another module. Each module using the process interaction approach, must call `receiveMessage` regularly. If it does not, the simulation will not proceed anymore. The purpose of most simulations is gathering statistical data about the algorithms simulated. For this purpose `cSimpleModule` offers the `finish()` method. This method is called at the end of a simulation run, before the module's corresponding objects are destroyed. It provides the module developer with the opportunity to gather all relevant data and for instance write it into a file.

5. REFLEX FOR OMNET++

A lot of research results in Wireless Sensor Networks (WSN) have been published which were solely based on data gathered in simulations and never have been verified with data from experiments on real hardware. Recent results suggest that simulation alone is not sufficient for evaluation of WSN algorithms. Due to the inherent complexity of distributed systems combined with uncertainties and fluctuation introduced with the wireless medium, simulators alone are incapable of providing sufficiently exact models. Nevertheless simulation is a vital step in the development process of WSN algorithms, since WSN deployment is tremendously expensive and debugging of embedded applications is cumbersome at best.

The goal of integrating Reflex into *OMNeT++* was therefore to provide researchers with the opportunity of implementing their algorithms once for the Reflex operating system and have that code run in a simulated WSN for test and basic evaluation as well as on real sensor nodes like TMoteSky or ScatterWeb with only minor porting effort.

When adding functionality to *OMNeT++* modules, the developer has the choice between two alternative programming models: *coroutine based* and *event-processing-function*. When using *coroutine-based programming* , the module code runs in its own thread. This thread is scheduled non-preemptively. It receives control from the simulation kernel each time the module receives an event (i.e. a message). Typically the function containing the coroutine code will never return. It will usually consist of an infinite loop containing send and receive calls. When using the *event-processing function*, the simulation kernel simply calls the given function of the module object with the message as argument. This function has to provide a run to completion semantic, i.e. it has to return immediately after processing the message. The drawback of the *coroutine-based* approach is that it requires more memory since every module needs its own CPU stack. When the model contains a large number of modules this can have significant influence on the simulation performance and its general applicability. On the other hand this approach is more *natural* if a module represents a sensor node.

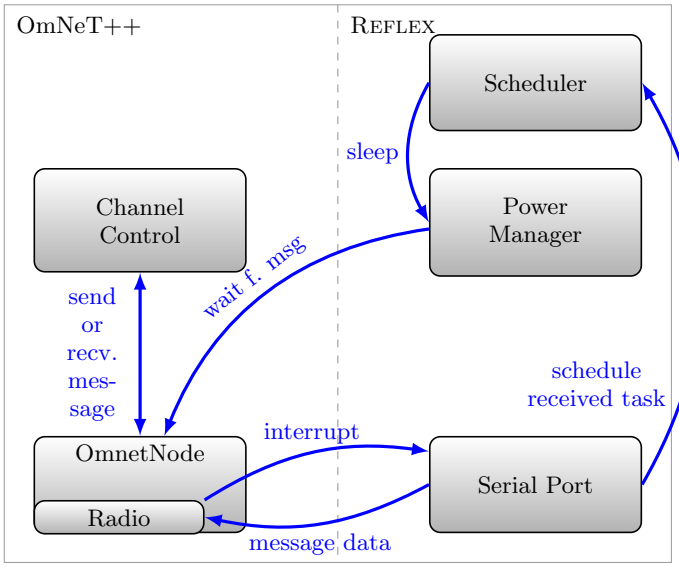

Figure 3: Components of the *OMNeT++* -version of Reflex

For our purpose the latter approach provides a less intrusive way of integrating Reflex. Since it is a single threaded system, Reflex can be easily mapped onto the coroutine model without modification. Although Reflex enforces run to completion semantics for tasks, it is still possible to use blocking code in the main thread (e.g. while waiting for a message to arrive). Such Reflex applications could not be run in the simulator when using the *event-processing-function*. For these reasons we decided to use the *coroutine-based* approach first, although it might be interesting to investigate the alternative, because it could provide better scalability regarding the number of sensor nodes a simulation model can contain, without suffering severe performance penalties. The limiting factor for the coroutine based approach is the total of memory available. Since each coroutine holds its own stack, an upper bound for the number of nodes in a simulation run is the amount of memory available divided by the amount of memory reserved for a coroutine stack. The latter should be chosen carefully, because if it was dimensioned too small, stack overflows will occur, resulting in unpredictable behavior of the simulation.

With the coroutine based approach there is one additional pitfall, which has to be kept in mind by the developer. Because the Reflex-scheduler continuously queries the task queue and only calls the wait routine when the queue is empty, the application must be designed such that there is no task which will be scheduled continuously without depending on external input. An application in which such a task exists, will effectively cause the simulation to stop

since it will never allow another *OMNeT++*-coroutine to be scheduled which results in no events being generated and so no time progress in the simulation being made.

5.1 Architecture

In figure 3 you can see the components involved in the integration of the REFLEX operating system into the *OMNeT++* simulator. The most central component is the *Omnet Node*, it serves as a gateway between the *OMNeT++* runtime environment and that of REFLEX. Viewed from an *OMNeT++* perspective, *Omnet Node* is a *SimpleModule*. The *Omnet Node* processes messages received from other nodes and initiates an interrupt handling. Whenever *Omnet Node* receives a message from *Channel Control* it reads the message's kind attribute and calls the interrupt guardian of REFLEX to indicate that there is an event pending at the *Serial Port*[1]. Therefore messages have to hold a valid interrupt vector in their kind attribute. *Channel Control* is the module which represents the wireless channel in the *INET* framework.

This approach allows adding additional devices to the simulation by simply defining a special message type representing an interrupt from that device. If a temperature sensor shall be modeled for instance, a message type, which contains the time and the measured temperature at that time would have to be created. At runtime the driver for the simulated sensor could read the appropriate values from a file containing data from a real measurement to fill in the fields of the message. The interrupt would be initiated by simply sending this message. Until now only one device in addition to the radio is implemented: the system timer.

A WSN typically consists of a greater number of nodes. Since an *OMNeT++* model is a monolithic application, all instances of *Omnet Node* reside in the same address space. Since each *Omnet Node* represents a REFLEX driven node, there are multiple instances of REFLEX running in a single address space. The REFLEX system needs reference to its main system object. This is usually no problem, since in one address space normally only one instance of REFLEX exists. Therefore a mechanism is needed which yields a reference to the system object of the currently active node. This is implemented via a global pointer, which is always set appropriately when control is transfered to another *Omnet Node* module.

Figure 4 visualizes the control flow of a message being received and processed. The module *Channel Control* models the wireless channel (it is part of the INET framework). It is responsible for calculating the set of receiving nodes for each message sent and delivering it accordingly. When the message is received by an *Omnet Node* its *kind* attribute is retrieved, which has to represent a valid entry in the interrupt vector table (see above). With this entry the module managing the vector table called *Interrupt Guardian* is invoked. The *Interrupt Guardian* invokes the appropriate handler. The handler, after doing some preprocessing *very* shortly, triggers an activity responsible for post processing. The trigger method of the activity then simply adds itself to the ready list of the scheduler, which will eventually activate it.

To understand how the transfer from the simulation engine to the REFLEX runtime and vice versa is realized, first the workflow of the REFLEX scheduler has to be considered. A scheduler mainly consists of an infinite loop in which the task queue is continuously checked for tasks. When it is not empty, the scheduler removes one task from the queue according to its scheduling strategy and activates it. When the task completes, the next task is selected and activated. When the list is empty, the scheduler signals the system that it may enter a power save state. The available power save modes are platform dependent. Therefore the corresponding method has to implemented for each platform. Recall that the portability of REFLEX is provided by strictly separating independent code from such depending on a certain platform resp. controller (refer to Section 3. The Linux controller can be used, since *OMNeT++* will run on Linux and most Unix based systems. The *OMNeT++* simulator then constitutes an additional platform. The code for this platform contains the *Omnet Node* module and some additional modules useful for a WSN model. It also contains a modified version of the `_wait()` system call which is called by the scheduler when the ready list is empty. Here the switch to the simulation runtime is implemented. The *OMNeT++* runtime is queried for new events (by issuing a `receive()` call), which may lead to a context switch to another coroutine.

```
_wait()
{
  //blocks until a message is handed
  //to this component
  active_message = receive();

  //interrupt handler
  dispatchMsg(msg);

  //return to scheduler
  return;
}
```

Figure 5: The code of the OMneT++ specific version of the `_wait()` method

In figure 5 the code for this method is shown. The public static member `active_node` of the class `OmnetNode` is necessary to give the handlers access to the message representing the interrupt. Since this information usually is available from special hardware registers or similar, the interface of the handlers does not provide means to submit it. The `dispatchMsg()` call reads the message contents, extracts the message kind carrying the index for the vector table and calls the interrupt resolving component of REFLEX with this index.

5.2 Experiences

Our simulation framework for REFLEX based WSN has for instance been used for simulating a combined routing layer for wireless sensor networks and mobile ad-hoc networks [17]. This example is particularly interesting because it demonstrates the possibility to integrate arbitrary models with our REFLEX based sensor node models. The object of the routing protocol was to support a scenario, where first responders equipped with some PDAs capable of WLAN and ZigBee communication are sent to a disaster area where a wireless sensor network that monitors the area for lethal

[1] we chose this name, because the transceiver is connected via the serial port on many sensor nodes

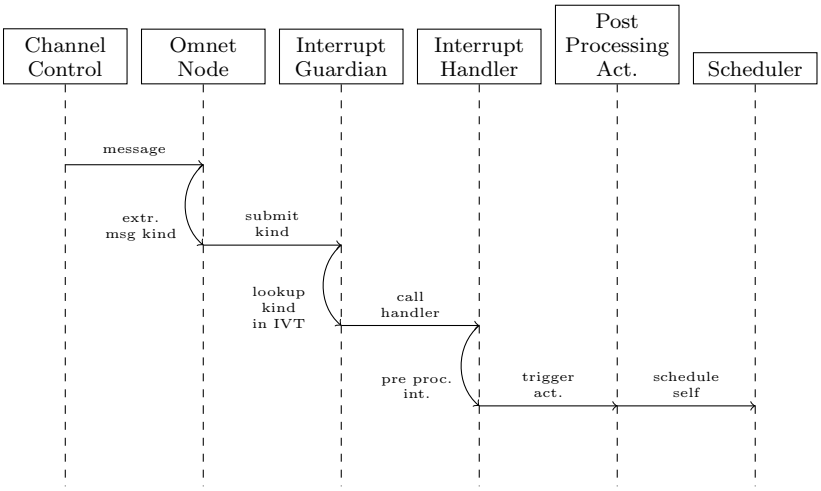

Figure 4: Interaction of OMneT++ and REFLEX components

amounts of contaminants or signs of life, has already been deployed. The idea is that the PDAs build a mobile ad-hoc network for mutual coordination of the first responders. Via ZigBee they are also capable of communicating with sensor nodes in range and therefore with the sensor network. The routing protocol should provide means to allow all combinations of mutual communication between PDAs and sensor nodes. The routing protocol should use weights assignable by the user to in route calculations. So it would for instance be possible, to use the MANET as a backbone for the WSN, conserving the small energy reserves of the sensor nodes.

For the evaluation besides experiments on real hardware, some simulations where executed. The simulated network consisted of 40 sensor nodes and 8 PDAs. The sensor nodes were modeled using the *OMNeT++* platform for REFLEX. The PDAs were modeled using CsharpSimpleModule [12], which allows definition of *OMNeT++*-modules in C#. For mobility and communication the INET framework has been used. The simulations were caried out on a PC with Intel XEON CPUs at 2.4 GHz nominal CPU speed and 2GB of RAM.

We recently used our framework for simulating up to 1000 nodes on a 10000 m * 10000 m playground. Nodes were communicating over a wireless channel using INETs *Channelcontrol* which we modified to simulate perfect links (i.e. no collisions and no bit errors). The communication pattern was rather sparse. To simulate the system clocks, a timer signal occured each millisecond at each node. We let the same simulations run on TOSSIM in parallel. Though we did not perform the simulations on a strictly comparable basis (different platforms, uncontrolled environmental influences) the results suggest that we are at least not slower than TOSSIM. The simulation results are presented in a paper currently under review so we can give no reference here. Our simulations were run on a MacBook with a 2.1 GHz Intel Core 2 Duo and 4 GB of RAM. Apart from setting the stack limit to 64 KB we did not need to make any changes to the systems settings.

This shows that our approach is capable of simulating networks of relevant size and it supports our claim that REFLEX models can be integrated with existing models.

6. CONCLUSION AND FUTURE WORK

We have presented the integration of an embedded operating system called REFLEX into the discrete event simulator *OMNeT++* . The goal was to provide a simulator for WSN, which enables developers to run applications written for REFLEX once in the simulator as well as on any hardware platform supported by reflex with only minor modifications. The *OMNeT++* platform provides radio communication based on the INET framework and a system timer to provide access to simulation time from the application. By utilizing some similarities in the design of REFLEX and *OMNeT++* seamless integration was easily possible.

We chose to use the coroutine based approach of *OMNeT++* , because that allowed for an integration with less modifications of REFLEX and imposes less restrictions on the developer regarding the program structure. Nevertheless it would be interesting to investigate the possibilities of the event processing approach. The benefit of that approach is that it would allow for larger simulations (i.e. with a greater number of nodes) since it induces much less runtime overhead than coroutines. To achieve this, the scheduler of REFLEX has to be reimplemented without the endless loop. The implementation would have to guarantee that every time the system receives a message the scheduler will be called after that message is handled.

7. REFERENCES

[1] Castalia. `http://castalia.npc.nicta.com.au/`.
[2] The inet framework. `http://www.omnetpp.org/staticpages/index.php?page=20041019113420757`.
[3] The mobility framework. `http://mobility-fw.sourceforge.net/`.
[4] Nesct: A language translator. `http://nesct.sourceforge.net/`.
[5] The network simulator. `http://www.isi.edu/nsnam/ns/`.
[6] The pawis simulation framework. `http://pawis.sourceforge.net/`.
[7] D. Braginsky and D. Estrin. Rumor routing algorthim for sensor networks. In *WSNA '02: Proceedings of the*

1st ACM international workshop on Wireless sensor networks and applications, pages 22–31, New York, NY, USA, 2002. ACM.

[8] E. Cheong, J. Liebman, J. Liu, and F. Zhao. Tinygals: a programming model for event-driven embedded systems. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 698–704, New York, NY, USA, 2003. ACM.

[9] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, Florida, USA, Nov. 2004.

[10] J. Eriksson, A. Dunkels, N. Finne, F. Österlind, and T. Voigt. Mspsim – an extensible simulator for msp430-equipped sensor boards. In *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session*, Delft, The Netherlands, Jan. 2007.

[11] K.Walther and J.Nolte. Event-flow and synchronization in single threaded systems. In *First GI/ITG Workshop on Non-Functional Properties of Embedded Systems (NFPES)*, 2006.

[12] A. Lagemann and J. Nolte. Csharpsimplemodule: writing omnet++ modules with c# and mono. In *Simutools '08: Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, pages 1–8, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[13] P. Levis, N. Lee, M. Welsh, and D. Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 126–137, New York, NY, USA, 2003. ACM.

[14] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. Tinyos: An operating system for sensor networks. *Ambient Intelligence*, pages 115–148, 2005.

[15] F. Österlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-level simulation in cooja. In *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session*, Delft, The Netherlands, Jan. 2007.

[16] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. Ght: a geographic hash table for data-centric storage. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 78–87, New York, NY, USA, 2002. ACM.

[17] T. Senner, R. Karnapke, A. Lagemann, and J. Nolte. A combined routing layer for wireless sensor networks and mobile ad-hoc networks. *Sensor Technologies and Applications, International Conference on*, 0:147–153, 2008.

[18] B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, page 67, Piscataway, NJ, USA, 2005. IEEE Press.

[19] A. Varga. The omnet++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference (ESM'2001)*, Prague, Czech Republic, June 2001. IEEE.

[20] K. Walther and J. Nolte. A flexible scheduling framework for deeply embedded systems. In *AINAW '07: Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops*, pages 784–791, Washington, DC, USA, 2007. IEEE Computer Society.

[21] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 31(3):9–18, 2002.