Grant agreement for:
Collaborative project, Small and medium-scale focused research project (STREP)

Deliverable D4.1:
Software Implementation and Integration Guidelines

Project acronym: FRONTS
Project full title: Foundations of Adaptive Networked Societies of Tiny Artefacts
Grant agreement no.: 215270

Responsible Partner: TUBS, UZL
Report Preparation Date: Monday, 4 August, 2008
Contents

1 Introduction 3

2 Fundamentals 5
   2.1 Shawn: A Customizable Sensor Network Simulator .......... 5
   2.2 iSense: A Modular Hardware and Software Platform for Wireless Sensor Networks ............................... 31

3 Development Environment Setup and Operation 37
   3.1 Shawn Setup ................................................. 37
   3.2 Shawn Operation ............................................. 51
   3.3 Application development with iSense .......................... 93

4 Coding Style 124
   4.1 General Style .................................................. 124
   4.2 Naming .......................................................... 127
   4.3 Programming Behavior ......................................... 128
   4.4 OOP .............................................................. 129
   4.5 Comments ....................................................... 129

5 Configuration Management 131
   5.1 Status Accounting ............................................. 132
   5.2 Version Control ................................................ 133
   5.3 Issue Tracking system ......................................... 136
   5.4 Integration of Components .................................... 137

6 Software Evaluation 141
   6.1 Correctness ..................................................... 141
   6.2 Efficiency ...................................................... 141
   6.3 Usability ....................................................... 142
   6.4 Maintainability ................................................ 142
   6.5 Verifiability .................................................... 143
   6.6 Re-usability .................................................... 143
   6.7 Interoperability ............................................... 143
   6.8 Expandability .................................................. 144

Bibliography 145
1 Introduction

This report represents the ”Software Implementation and Integration Guidelines” document as Deliverable 4.1 of the project FRONTS - Foundations of Adaptive Networked Societies of Tiny Artefacts. It contains the topics Coding Style, Development Environment Setup and Operation, Configuration Management and Software Evaluation.

The document is structured as follows.

Chapter 2 contains descriptions about the fundamentals of Shawn and iSense. Chapter 2.1 gives an introduction to Shawn, a customizable sensor network simulator. Amongst others the design goals of Shawn and the realization of Shawn are described here. Chapter 2.2 deals with iSense, a modular hardware and software platform for wireless sensor networks. The fundamentals of available hardware and software and their modularity are explained at this point. Both technologies – Shawn and iSense – will be used in the FRONTS project to provide a central experiments repository and experimentation testbed. Repository and testbed provide common platforms for the project’s participants to test algorithms, protocols, etc. that will be implemented within the project. That way uniformity, reusability and the quality of the developed software shall be guaranteed.

Chapter 3 describes the development environment setup and operation. Detailed install and getting started instructions are given here. The Chapter is splitted into three parts, two about simulating applications for wireless sensor networks with Shawn. Section 3.1 deals with the installation of Shawn while Section 3.2 deals with the operation of Shawn. The third Section gives instructions for using iSense to also be able to transfer implemented applications on real hardware (Section 3.3). The part about iSense is again divided into three Sections. Section 3.3 explains how to combine Shawn and iSense so that it is not just possible to simulate applications with Shawn only but also how to write applications for iSense and compile for and use them with Shawn. Section 3.3 describes the available iSense hardware and illustrates how to program the available devices. Section 3.3 finally describes the FRONTS testbed that will be built up in Lübeck on the one hand and the software that is needed to enable remote access to the testbed by the project partners on the other hand.

Chapter 4 specifies the desired coding style for developing Shawn- and iSense-source code.

In Chapter 5 the configuration management policies are specified.

Finally Chapter 6 describes the quality factors that are to pay regard to at the software development by developers of the FRONTS project to achieve quality assurance of the project’s results.

For a quick view at special install or development guides for Shawn please have a look at Table 1 for page references.
If you are interested in special install or development guides for *iSense* only you can find corresponding page references in Table 2.

Table 2: Document’s page references for *iSense*

Table 3 gives page references to the topics *Fundamentals, Development Environment Setup and Operation, Coding Style, Configuration Management* and *Software Evaluation*.

Table 3: Document’s page references to topics

**Acknowledgements**

We would like to thank coalesenses GmbH [coa06] for giving us support concerning questions about the most efficient use of the iSense hardware and software. Some text and pictures in this document were also provided by courtesy of coalesenses GmbH.
2 Fundamentals

2.1 Shawn: A Customizable Sensor Network Simulator

Software for Wireless Sensor Networks (WSNs) must be thoroughly tested prior to real-world deployments since sensor nodes do not offer convenient debugging interfaces and are typically inaccessible after deployment. Furthermore, successfully designing algorithms and protocols for WSNs requires a deep understanding of these complex distributed networks. To achieve these goals, three different approaches are commonly used:

- Analytical methods
- Real-world experiments
- Computer simulations

Analytical methods are typically not well-suited to support the development of complete WSN applications. Despite their expressiveness and generality, it is difficult to grasp all details of such complex, distributed applications in a purely formal manner.

Real-world experiments are an attractive option as they are a convincing means to demonstrate that an application is able to accomplish a specific task in practice – if the technology is already available. However, due to the unpredictable environmental influences it is hard to reproduce results or to isolate sources of errors. Furthermore, real-world deployments are laborious and involve management tasks that are not directly related to the problem [HLP+06]. For this reason, they are typically limited to a few dozens of devices [SMP+04, WTV+07], while future scenarios anticipate networks of several thousands to millions of nodes [EGH00, Kum03].

Computer simulations are a promising means to tackle the task of algorithm and protocol engineering for WSNs. A number of simulation tools are available. They reproduce real-world effects inside a simulation environment including radio propagation properties and environmental influences. This mitigates required efforts for real-world deployments and may therefore help to increase their size. However, the high level of detail provided by these tools obfuscates and misses another, much more crucial issue: The large number of factors that influence the behavior of the whole network renders it nearly impossible to isolate a specific parameter of interest.

For example, consider the development of a novel routing protocol. In the case of a very low throughput, the cause of the problem is not clear at first sight, as the sources for the error are manifold: the MAC layer might be faulty; cross-traffic from other senders could limit the available bandwidth; radio propagation properties might have changed or the routing protocol’s algorithm is not yet optimal. Therefore, it is not only sufficient to simulate a high number of nodes with a high level of detail.
Instead, developers require the ability to focus on the actual research problem. When designing algorithms and protocols for WSN it is important to understand the underlying structure of the network – a task that is often one level above the technical details of individual nodes and low-level effects.

There is certainly some influence of communication characteristics, e.g., because they affect transmission times, communication paths and packet loss. From the algorithm’s point of view, there is no difference between a complete simulation of the physical environment (or low-level networking protocols) and the alternative approach of using well-chosen random distributions on message delay and loss. Thus, using a detailed simulation may lead to the situation where the simulator spends much processing time on producing results that are of no interest at all. By contrast, they actually hinder productive research on the algorithm.

To improve this situation, we propose a novel simulation tool called Shawn [KPB+05, PFKF05, FKFP07]. The central idea of Shawn is to replace low-level effects with abstract and exchangeable models so that simulations can be used for huge networks in reasonable time while keeping the focus on the actual research problem. In the following, Section 2.1 summarizes related work. Then, Section 2.1 discusses fundamental design goals of Shawn while Section 2.1 shows how these goals reflect themselves in Shawn’s architecture. Finally, Section 2.1 compares Shawn’s performance with two other prominent simulation tools (Ns-2 and TOSSIM).

Related Work

The range of applications for simulations is rather broad. Consequently, many different simulation tools have been developed. Each of them targets a specific application domain where it delivers best results. In the following, simulation tools frequently used in WSN research are presented. Figure 1 provides an overview of these tools and classifies their application area along two axes: abstraction level and the typical network sizes. Note that this does not express the maximal feasible network sizes, but rather reflects typical application domains.

Ns-2

The Network Simulator-2 (Ns-2, [Uni]) is a discrete event simulator targeted at network research. Nowadays, Ns-2 is the most prominent network simulator used in WSN research [KCMC05]. It focuses on the simulation of ISO/OSI layers including energy consumption and phenomena on the physical layer. Ns-2 includes a vast repository of protocols, traffic generators and tools to simulate TCP, routing and multicast protocols over wired and wireless networks. It features detailed simulation tracing and includes the visualization tool network animator (nam) for later playback of the observed traffic. Support for sensor network simulations has also been integrated [Nav, PSS00], including sensor models, battery models, lightweight protocol
stacks and scenario generation tools.

The highly detailed packet level simulations lead to a runtime behavior closely coupled with the number of packets being exchanged, making it nearly impossible to simulate large networks. Ns-2 is capable of handling up to 16,000 nodes but the detail level of its simulations render working with more than 1,000 nodes virtually impossible in terms of runtime and memory consumption.

**OMNeT++** The *Objective Modular Network Testbed in C++* (OMNeT++, [Var]) is an object-oriented, modular discrete event simulator. It is very similar to Ns-2 and also targets the ISO/OSI model. It can handle a few thousands of nodes and features a graphical network editor as well as a visualizer for the network and the data flow. The simulator is written in C++ and comes with a homegrown configuration language called NED. OMNeT’s main objective is to provide a component architecture through which simulations can be composed very flexible. Components are programmed in C++ and then assembled into larger components using NED. It is free for academic use only and a commercial license is available.

**SENSE** The *Sensor Network Simulator and Emulator* (SENSE, [SCBZ]) is a simulator specifically developed for the simulation of sensor networks. The authors mention extensibility and reusability as the key factors they address with SENSE. Extensibility is tackled by avoiding a tight coupling of objects through a *component-port* model, which removes the interdependency of objects that is often found in object-oriented architectures. This is achieved by their proposed simulation component classifications, which are essentially interfaces, enabling the exchange of implementations without the need to change the actual code. SENSE offers different battery models, simple network and application layers and an IEEE 802.11 [IEEa]
implementation. In its current version, it provides a sequential simulation engine that can cope with around 5,000 nodes. Depending on the communication pattern of the network, this number may drop to 500. The authors plan to support parallelization of the simulations to increase the overall performance.

**TOSSIM**  The *TinyOS mote simulator* (TOSSIM, [LLWC03]) emulates TinyOS [LMP+05, The05, HSW+00] motes down to the bit level and is hence a platform specific simulator. It compiles code written for TinyOS to an executable file that can be run on standard PC equipment. It ships with a GUI (TinyViz), which can visualize and interact with running simulations. Recently, PowerTOSSIM [SHC+04], a power modeling extension has been integrated into TOSSIM. PowerTOSSIM models the power consumed by TinyOS applications and includes a detailed model of the power consumption of the Mica2 [Cro] motes. Using this technique, developers can test TinyOS applications without deploying them on real sensor network hardware. TOSSIM can handle scenarios with a few thousand virtual TinyOS nodes.

The crucial point of the above presented simulation tools is that each of them has its area of expertise in which it excels. Unfortunately, none of these areas happens to be high-level protocols and abstract algorithms in combination with the speed to handle large networks. This gap is filled by Shawn.

**Design Goals**

Shawn differs in various ways from the above-mentioned simulation tools, while the most notable difference is its focus. It does not compete with these simulators in the area of network stack simulation. Instead, Shawn emerged from an algorithmic background. Its primary design goals are:

- Simulate the effect caused by a phenomenon, not the phenomenon itself.
- Scalability and support for extremely large networks.
- Free choice of the implementation model.

**Simulate the effects**  As discussed in Section 2.1, most simulation tools perform a complete simulation of the MAC layer including radio propagation properties such as attenuation, collision, fading and multi-path propagation. A central design guideline of Shawn is to *simulate the effect caused by a phenomenon, and not the phenomenon itself*. Shawn therefore only models the effects of a MAC layer for the application (e.g., packet loss, corruption and delay).

This has several implications on the simulations performed with Shawn. On the one hand, they are more predictable and there is a performance gain since such a model can be implemented very efficiently. On the other hand, this means that Shawn is unable to provide the same detail level that, for example, Ns-2 provides with regard
to physical layer or packet level phenomena. However, if the model is chosen well, the effects for the application are virtually the same. Imagine two implementations of a MAC layer: One abstract implementation that yields an increased packet loss on high local traffic and one that calculates interference for single packets using radio propagation models. Both will produce similar effects on the application layer.

It must be mentioned though that the interpretation of obtained results must take the properties of the individual models into account. If, for instance, a simplified communication model is used to benchmark the results of a localization algorithm, the quality of the obtained solution remains unaffected. However, the actual running time of the algorithm is not representative for real-world environments since no delay or loss occurs.

**Scalability** One central goal of Shawn is to support orders of magnitudes higher numbers of nodes than the currently existing simulators. By simplifying the structure of several low-level parameters, their time-consuming computation can be replaced by fast substitutes as long as the interest in the large-scale behavior of the system focuses on unaffected properties. A direct benefit of this paradigm is that Shawn can simulate vast networks.

To enable a fast simulation of these scenarios, Shawn can be custom-tailored to the problem at hand by selecting appropriate configuration options. This enables developers to optimize the performance of Shawn specifically for each single scenario. For example, a scenario without any mobility can be treated differently than a scenario where sensor nodes are moving.

**Free model choice** Shawn supports a multi-stage development cycle where developers can freely choose the implementation model as depicted in Figure 2. Using Shawn, they are not limited to the implementation of distributed protocols. The rationale behind this approach is that – given a first idea for a novel algorithm – the next natural step is not the design of a fully distributed protocol. In fact, it is more likely to perform a structural analysis of the problem at hand. To get a better understanding of the problem in this phase, it may be helpful to look at some exemplary networks to analyze their structure and the underlying graph representation.

The next step may be to implement a centralized version of the algorithm in order to achieve a rapid prototype version. A centralized algorithm has full access to all nodes and has a global, flat view of the network. This provides a simple means to obtain results and get a first impression of the overall performance of the algorithm in question. The results emerging from this process can provide optimization feedback for the algorithm design.

Once a satisfactory state of the centralized version has been achieved, the feasibility of its distributed implementation can be investigated. Since the aim of this step
is to prove that the algorithm can be transformed to a distributed implementation, a simplified communication model between individual sensor nodes can be utilized. This allows for an efficient and fast implementation, yet with meaningful results.

This simplified version can be transformed gradually into a protocol that actually exchanges network messages containing protocol payload. With the protocol and data structures in place, the performance of the distributed implementation can be evaluated. Interesting questions that can be explored are for instance the number of messages, energy consumption, run-time, resilience to message loss and effects of the environment. This development cycle helps the developer to start from an initial idea and gradually leads to a fully distributed protocol. However, each step of the cycle is optional and it is up to the developer to pick only the necessary ones.

**Architecture**

Shawn’s architecture comprises three major parts (cp. Figure 3):

- **Models**
- **Sequencer**
- **Simulation Environment**

Every aspect of Shawn is influenced by one or more Models, which are the key to its flexibility and scalability. The Sequencer is the central coordinating unit in Shawn as it configures the simulation, executes tasks sequentially and controls the simulation. The Simulation Environment is the home for the virtual world in which the simulated sensor nodes reside. In the following, the functionality of these core components is described in detail.
Figure 3: High-level architecture of Shawn and overview of its core components
**Models**  Shawn distinguishes between *models* and their respective implementations. A model is the interface used by Shawn to control the simulation without any knowledge on how a specific implementation may look like. Shawn maintains a repository of model implementations that can be used to compose simulation setups by selecting the desired behaviors. The implementation of a model may be simplified and fast, or it could provide close approximations to reality. This enables the user to select the most appropriate implementation of each model to fine-tune Shawn’s behavior for a particular simulation task.

As depicted in Figure 3, three models form the foundation of Shawn:

- *Communication Model*
- *Edge Model*
- *Transmission Model*

In the following, these models and their already included implementations are explained in detail. Other models of minor importance are briefly introduced at the end of this section.

**Communication Model**  Whenever a simulated sensor node in Shawn transmits a message, the potential receivers of this message must be identified by the simulator. Please note that this does not determine the properties of individual transmissions but defines whether two nodes can communicate as a matter of principle. This question is answered by implementations of the *Communication Model*. Figure 4 presents the C++ interface of the *Communication Model*. A single method is invoked to determine whether the node \( b \) is in reach of the node \( a \).

```cpp
class CommunicationModel {
    ...
    bool can_communicate_uni (Node& a, Node& b);
    ...
};
```

Figure 4: Application programming interface of the communication model in Shawn (excerpt)

By implementing this interface with user-defined code, arbitrary communication patterns can be realized. Shawn ships with a set of different *Communication Model* implementations that are shown in Figure 5.

Three of these five implementations resemble communication patterns that are often used in WSN research. Figure 6 shows examples of how these models work. In the figure, the shared neighbors (filled black circles) of two nodes \( n_1 \) and \( n_2 \) (filled black circle with an extra black ring) are highlighted.

The *Unit Disk Graph* (UDG, cp. Figure 6(a)) radio model is based on the observation that the signal strength fades with the square of the distance from the sender.
Figure 5: Overview of Shawn’s communication models

Figure 6: Characteristics of different radio models
Given a minimum signal strength required for reception, two nodes can communicate bidirectional if the Euclidean distance $d$ between the nodes is less than $r_{\text{max}}$. Regardless of its substantial abstractions from the real world, the model is widely utilized in WSN research because of its simplicity.

The Quasi-Unit Disk Graph (Q-UDG) radio model is a variant of the model introduced in [BFN01]. It defines two new distances $r_1$ and $r_2$ with $r_1 < r_2$. For $0 < d < r_1$ and $d > r_2$, the behavior is equivalent to the UDG Model. For $r_1 \leq d \leq r_2$, the packet reception probability decreases linearly from 1 to 0. It therefore honors the fact that the probability of a successful reception diminishes with increasing distance. Figure 6(b) shows an example with $r_1 = 0.75 \ast r_{\text{max}}$ and $r_2 = 1.25 \ast r_{\text{max}}$.

Based on real-world experiments, the Radio Irregularity Model (RIM, [GKW+02, ZHKS04, ZHKS06]) proposes an angle dependant range between a minimum and a maximum communication range ($r_{\text{min}}$ and $r_{\text{max}}$). A factor determines the maximum change in transmission range per degree and thus controls the irregularity of the shape (cp. Figure 6(c)). In contrast to UDG and Q-UDG, the RIM model also yields unidirectional links.

In addition to the real-world models, the Permanent Link model allows the specification of static links to pre-define communication relations such as wired connections to gateway nodes. The Chained Communication Model supports combining multiple communication models to a single communication model. For instance, while most of the sensor nodes in a network could use the Unit Disk Graph model, some gateway nodes have wired connections that are modeled by a Permanent Link model.

**Edge Model** The Edge Model provides a graph representation of the network. The simulated nodes are the vertices of the graph and an edge between two nodes is added whenever the Communication Model returns true. To assemble this graph representation, the Edge Model repeatedly queries the Communication Model. It is therefore possible to access the direct neighbors of a node, the neighbors of the neighbors, and so on. This is used by Shawn to determine the potential recipients of a message by iterating over the neighbors of the sending node. Simple centralized algorithms that need information on the communication graph can be implemented very efficiently as in contrast to other simulation tools, no messages must be exchanged to access the direct neighbors of a node. Simple centralized algorithms that need information on the communication graph can be implemented very efficiently as in contrast to other simulation tools, no messages must be exchanged to access the direct neighbors of a node.

Figure 7 depicts the C++ interface that must be extended by implementations of the Edge Model. In essence, two methods provide the ability to iterate over the neighbors of a node for a specific communication direction. The communication direction parameter defines how the property “neighbor” is interpreted and can be one of the following: in, out, bidirectional or any. If not specified, the communication direction defaults to “bidi”. 

---

215270 (FRONTS) – Deliverable D4.1
The communication direction property is defined as follows: \( u \) and \( v \) are neighbors if for the communication direction

- “in” holds: \( \text{can\_communicate\_uni}(v, u) \),
- “out” holds: \( \text{can\_communicate\_uni}(u, v) \),
- “bidi” holds: \( \text{can\_communicate\_uni}(u, v) \land \text{can\_communicate\_uni}(v, u) \),
- “any” holds: \( \text{can\_communicate\_uni}(u, v) \lor \text{can\_communicate\_uni}(v, u) \).

Depending on the application’s requirements and its properties, different storage models for these graphs are needed. For instance, mobile scenarios require different storage models than static scenarios. In addition, simulations of relatively small networks may allow storing the complete neighborhood of each node in memory. Conversely, huge networks will impose impractical demands for memory and hence supplementary edge models trade memory for runtime, e.g., by recalculating the neighborhood on each request or by caching a certain amount of neighborhoods. Accordingly, Shawn provides different \textit{EdgeModel} implementations as shown in Figure 8.

![Figure 8: Overview of Shawn’s edge models](image)

The \textit{Lazy} edge model is intended for simulations with only a small amount of nodes, hardly any communication or high node mobility. It does not store any infor-
The Grid edge model uses a two-dimensional grid for arranging nodes according to their geometric position. Therefore, the search for neighboring nodes is restricted to nearby ones, thus effectively improving the lookup speed while still fully supporting mobility. The List edge model stores the complete graph of the network in memory. This allows for a faster iteration over the neighboring nodes at the cost of a time-consuming initial construction and non-negligible memory demands. Since the list is only built once at the beginning of the simulation, this edge model does not support node mobility but only static scenarios. The Fast List edge model combines the functionality of the Grid and List edge models into a single model implementation. Internally, it uses a Grid edge model for the initial construction of a contained List edge model. As a result, it provides the features of the List edge model plus a fast construction at the cost of a higher initial memory footprint.

Transmission Model Whenever a node transmits a message, the behavior of the transmission channel may be completely different than for any other message transmitted earlier. For instance, cross traffic from other nodes may block the wireless channel or interference may degrade the channel’s quality. To model these transient characteristics inside Shawn, the Transmission Model determines the properties of an individual message transmission. It can arbitrarily delay, drop or alter messages. This means that a message may not reach its destination even if the Communication Model states that two nodes can communicate as a matter of principle and the Edge Model lists these two nodes as neighbors. Figure 9 shows the C++ interface for transmission model implementations in Shawn. The send_message() -method accepts a MessageInfo data structure containing the message itself, the time of transmission and the position of the sender.

```cpp
class TransmissionModel {
    struct MessageInfo {
        Node* src_; 
        Vec src_pos_; 
        double time_; 
        MessageHandle msg_; 
    }; 

    void send_message (MessageInfo&); 
};

Figure 9: Application programming interface of the transmission model in Shawn (excerpt)
```

Again, the choice of an implementation strongly depends on the simulation goal. In case that the runtime of an algorithm is not of interest but only its quality, a simple transmission model without delay, loss or message corruption is sufficient. Models
that are more sophisticated could take contention, transmission time and errors into account at the cost of performance.

![Figure 10: Overview of Shawn’s transmission models](image)

Figure 10 lists the built-in transmission models of Shawn, covering both abstract and close-to-reality implementations. The **Reliable** transmission model delivers all messages immediately, without loss or corruption to all neighboring nodes. **Random drop** is a slight variation in that it discards messages with a given probability but it neither delays nor alters messages. The **Statistics** implementation does not deliver any message but instead records informational data such as the overall message count, different message types, etc. To make use of such a non-functional transmission model, the **Chainable Transmission Model** allows a series of transmission models to process a message sequentially. Like that, a message could first be counted, then delayed and may then be dropped by combining several simple transmission models.

Two additional implementations are closer to the real world than the above-mentioned ones. They simulate the effects of the well-known CSMA/CA and (slotted) Aloha [Abr70, CDK05, TS01] medium access schemes. Please note that not the MAC protocol itself is simulated but only the delay and loss characteristics are modeled for performance reasons.

**Miscellaneous models** Besides these core models that shape the fundamental behavior of Shawn, a number of more specialized models provide data for the simulations. Currently, Shawn contains models for random variables, distance estimations and mobility.

Random variables are often needed in simulations to mimic uncertainty and randomness present in the real world. The **Random Variable** model introduces a layer of abstraction between the actual sources of random data and the application. As a result, algorithms can be tested with different underlying random variables with-
out changes to the implementation. Sensor nodes often need distance estimations to other nodes to acquire context information such as their position. In Shawn, the Node Distance Estimate model provides these distance estimations, e.g. to support the evaluation of localization algorithms. Modeling arbitrary mobility is supported by so-called Node Movements. Implementations of this model provide the current position of a node when queried, e.g. by the communication model or an application. In order to allow an observation of the movement of nodes, listeners can register with the movements for location updates. Whenever a node is about to leave a previously supplied area given by a bounding box, it notifies the listener and obtains a new bounding box. This mechanism is used e.g. by the Grid edge model to adapt its internal status.

**Sequencer** The sequencer is the control center of the simulation: it prepares the world in which the simulated nodes live, instantiates and parameterizes the implementations of the models as designated by the configuration input and controls the simulation. It consists of

- *Simulation Tasks*, the
- *Simulation Controller* and the
- *Event Scheduler*.

**Simulation Tasks** *Simulation Tasks* are pieces of code that are invoked from the configuration of the simulation supplied by the user. They are not directly related to the simulated application but they have access to the whole simulation environment and are thus able to perform a wide range of tasks. Example uses are managing simulations, gathering data from individual nodes or running centralized algorithms.

Shawn exclusively uses tasks to expose its internal features to the user. A variety of tasks is included in Shawn that supports the creation and parameterization of new simulation worlds, nodes, routing protocols, random variables, etc. Even the actual simulation is triggered using a task and the user can specify the amount of time that should be simulated upon the execution of this simulation task.

```cpp
class SimulationTask {

    void run (SimulationController&);
    string name ();
    string description ();
    ...
};
```

Figure 11: Application programming interface of a Simulation Task in Shawn (excerpt)

*Simulation Tasks* are configured and invoked by the *Simulation Controller* as discussed later on. They are identified and invoked using a unique name and parameters.
are passed as simple \((name, value)\)-pairs. Figure 11 presents the C++ interface that a simulation task implementation must extend: One method that returns the unique identifying name, one that returns a human readable description and one that performs the actual task.

**Simulation Controller** The *Simulation Controller* acts as the central repository for all available model implementations and runs the simulation by transforming the configuration input into parameterized invocations of *Simulation Tasks*. In doing so, it mediates between Shawn’s simulation kernel and the user. In line with most other components of Shawn, the *Simulation Controller* can be customized by a developer to realize an arbitrary control over the simulation. The default implementation reads the configuration commands from a text file or the standard input stream. Figure 12 shows how commands are structured.

Two different line formats can occur: one that defines global variables and one that invokes and parameterizes Simulation Tasks. Line 1 shows how a global \((name, value)\)-pair is specified which is valid from the point of its definition until the simulation run completes. Line 3 shows how a task is invoked by specifying its name (as returned by the task’s \(name()\)-method, cp. Figure 11) separated with a trailing blank character. Following the tasks name, \((name, value)\)-pairs may occur that are valid for the invocation of the task (local values temporarily overwrite global ones if their names are identical).

If a simulation task with this name is found, its \(run()\)-method is invoked and the current set of \((name, value)\)-pairs is passed. Note that tasks are instantiated when Shawn starts and using the same task-name again results in invoking the \(run()\)-method on the same instance as for the first call. This can be used to collect data continuously and to evaluate it at a later point in time.

```
1. global_name1=global_value1
2. task_name name1=value1 name2=value2
```

Figure 12: Plain-text file format used by Shawn’s default Simulation Controller implementation

A second implementation allows the use of Java-language scripts to steer the simulation. While providing the same functionality, it allows more complex constructs and evaluations already in the configuration file.

**Event Scheduler** Shawn uses a discrete event scheduler to model time. The *Event Scheduler* is Shawn’s timekeeping instance. Objects that need the notion of time can register with the Event Scheduler to be notified at an arbitrary point in time. The simulation always skips to the next event time and notifies the registered
handlers. This process continues until all nodes signal either that they have powered down or until the maximum configured time has elapsed.

This has some performance advantages compared with traditional approaches that use fixed time intervals (such as a clock-tick every 1ms): First, handlers are notified only at the precise time that they have requested avoiding unnecessary calls to idle or waiting nodes that have no demand for processing. Second, users are not bound to some artificial granularity but an event may occur at the full precision that is offered by floating-point numbers. Figure 13 sketches how the Event Scheduler allows simulations to utilize this timing mechanism by using one of the several interaction possibilities.

![Figure 13: Shawn’s discrete event scheduler](image)

Shawn logically arranges simulations into rounds \((r = 0, 1, 2, \ldots)\). The user may register Simulation Tasks as pre-step and post-step tasks that are executed immediately before and after each of these rounds. This is useful to extract information from the simulation without the need to intermingle simulation code with code that analyzes the performance of the simulated algorithm. At the beginning of each round, a node’s `work()`-method is invoked. Applications can choose to use this method for an eased implementation when precise timing is not required. Apart from these fixed points in time, the nodes or other elements of the simulation may register an event at any point in time. Using these three distinct possibilities offers applications the required flexibility to integrate timing aspects into the simulation without degrading the overall performance.

**Simulation Environment** The simulation environment is the home of the virtual world in which the simulation objects reside. As shown in Figure 3, the simulated Nodes reside in a single World instance. The Nodes themselves serve as a container for so-called Processors. Developers using Shawn implement their application logic as instances of these Processors. By decoupling the application inside a Processor from the Node, multiple applications can easily be combined in a single simulation run without changing their implementations. For instance, one processor could implement an application specific protocol while another processor gathers statistics.
Figure 14 shows an excerpt of the Processor’s API. After a processor has been instantiated, its `boot()`-method is invoked. A Processor can transmit messages by a call to `send()` and whenever a message for the Node is received, it dispatches this message to all its Processors by calling `process_message()`. As mentioned above, the Processor’s `work()`-method is invoked whenever the Event Scheduler starts a new simulation round.

```cpp
class Processor
{
    ...
    void boot( );
    bool process_message( MessageHandle& );
    void work( );
    Node& owner( );
    void send( MessageHandle& );
    ...
};
```

A Node offers a number of services to the Processors that ease the implementation of algorithms and simplify the overall simulation task. As mentioned above, the Edge Model can be used to identify the neighbors of the current node. Unlike other simulators, Shawn does not restrict the user to a message-driven programming model but also allows “shortcuts”, i.e., to execute method calls directly on other nodes. This is beneficial when implementing a centralized version of an algorithm or to get things done fast that are only a pre-condition for the current simulation, but not in the focus of the research.

It is often the case that an algorithm requires input that is produced by multiple (potentially very complex) other algorithms. To avoid waiting for the same results of these previous steps repeatedly in every simulation run, Shawn offers the ability to attach type-safe information to Nodes, the World and the Simulation Environment. Two Simulation Tasks (`load_world` and `save_world`) provide the ability to load Tags from and to save Tags to XML documents. Currently, three different Tag types exist: Simple Tags, Group Tags and Map Tags. Simple Tags contain a value of a certain type (e.g., `string`, `int` or `Boolean`), Group Tags contain other Tags and Map Tags contain pairs of values of a certain type. Figure 15(a) shows such an XML document that contains these different Tag types and Figure 15(b) shows how a Processor would access the Boolean Tag called `base_station`.

A benefit of this concept is that it allows decoupling state variables from member variables in the user’s simulation code. By this means, parts of a potentially complicated protocol can be replaced without code modification because the internal state is stored in tags and not in member variables of a special implementation.

To model sensors and their corresponding sensor values, a generic framework called `Readings` and `Sensors` is provided. Readings deliver position-dependent and
time-dependent values that can be arbitrarily typed. Sensors are bound to a specific
sensor node and deliver sensor readings. Readings and sensors can be configured
by the user and are referenced inside the simulation using unique names. This de-
coupling allows changing the underlying reading or sensor implementation without
changing or recompiling the simulation code. In its current version, Shawn already
provides some simple sensor and reading implementations, which obtain their values
from XML files or Tags.

Evaluation

This section evaluates the performance and adaptability of Shawn. It first compares
Shawn with Ns-2 and TOSSIM to demonstrate that Shawn can handle large networks
at high speeds. Then Shawn’s adaptability and flexibility is demonstrated at the ex-
ample of the available Edge Models.

Comparison with Ns-2 and TOSSIM Since the exchange of wireless messages
is the key ingredient in wireless sensor networks, a simulator’s ability to dispatch
messages to their recipients determines the speed of simulations. In the following,
measurements are presented that show the amount of memory and CPU time required
to simulate a simple application that broadcasts a message every 250ms of simulated
time. The communication range of the sensor nodes is set to 50 length units and each simulation runs for 60 simulated time units. The size of the simulated area is 500x500 length units. A number of simulations with increasing node count were performed. Therefore, the network’s density increases steadily as more nodes are added to the scenario. This application has been implemented for Ns-2, TOSSIM and Shawn. All simulation tools were used as supplied by the source repositories with maximal compiler optimization enabled. The simulations were run on standard, state of the art i686 PCs.

![Graph showing CPU time vs. number of nodes for Ns-2, Shawn, and TOSSIM](image)

**Figure 16: Comparison of the required CPU time of Shawn, Ns-2 and TOSSIM**

Figure 16 depicts the required CPU time in seconds and Figure 17 shows the maximally used amount of RAM for the three simulation tools at different node counts. It should be noted that this kind of comparison is biased in favor of Shawn because the two other simulators perform much more detailed computations to arrive at the same results. It is more to be seen as an indication how application developers can benefit from using Shawn when these detailed results are not in the focus of interest. When interpreting the results, please note that these figures as well as the following ones use a logarithmic scale on both axes.

The first thing to notice is that Shawn outperforms both other simulation tools by orders of magnitude. Ns-2 hits the one-day barrier where Shawn is still finishing in less than one minute with a considerably smaller memory footprint. As mentioned
Figure 17: Comparison of the required amount of RAM of Shawn, Ns-2 and TOSSIM
above, this is because Ns-2 performs a very detailed simulation of lower layers such as the physical and the data link layer while Shawn simply dispatches the messages using a simplified model. Nearly the same situation applies to TOSSIM that simulates an underlying TinyOS-supported hardware platform. This clearly shows that Shawn excels in its area of expertise – the simulation of large-scale sensor networks with a focus on abstract, algorithmic considerations and high-level protocol development.

Adaptability of Shawn As discussed in Section 2.1, Shawn’s runtime behavior is heavily influenced by choosing a distinct implementation of one of the models (e.g., the Edge Model, Communication Model or the Transmission Model). Depending on the simulated scenario, a different choice may substantially alter the performance and the resource consumption of the simulation. Since selecting a specific implementation simply requires changing a value in the configuration file, users can select the best implementation for each simulation.

To demonstrate the pros and cons of the different edge model implementations, the above-described simulations were repeated using the same underlying scenarios using different edge models. Figure 18 shows the required CPU time for the different implementations and node counts. It is evident, that the List and the Fast List are faster than Grid and much faster than Simple. However, considering the memory consumption (cp. Figure 19), this performance gain comes at a certain price.

Because the size of the simulated world is fixed at 500x500 length-units with a communication range of 50 length-units, the density of the underlying graph – and therefore the amount of memory required for storing the neighbors of each node – increases significantly the more nodes are added to the scenario. A compromise between speed and memory consumption is offered by Grid.
Figure 18: Required CPU time of Shawn’s edge models (Constant size)
Figure 19: Memory consumption for Shawn’s edge models (Constant size)
However, this observation does not hold in general. Consider the situation where the number of nodes increases while the average density of the network remains constant. Figure 20 and Figure 21 depict the result of such simulations with a constant density (the density corresponds to the one of a scenario with 100 nodes in the simulations presented above).

![Figure 20: Required CPU time of Shawn’s edge models (Constant density)](image)

When looking at simulations using *Fast List*, the runtimes are an order of magnitude lower than in previous case. However, the memory requirements differ only marginally (maximal difference 108.64 MByte, 9.94 MByte in average). This is because, in contrast to the previous scenario, the neighborhood sizes remain constant and the storage space required for keeping them in memory increases only slowly. As in the previous example, *Simple* is the slowest one and *Fast List* the fastest edge model implementation.

In this example, *Grid* outperforms *List* only for node counts larger than 10,000. This is because of the high initial setup cost of the *list* edge model. However, *Fast List* performs better for all node counts which is because it reduces the time required for construction of a *List* edge model by using a *Grid* internally to limit the search space for neighboring nodes.
Figure 21: Memory consumption for Shawn’s edge models (Constant density)
**Conclusion**  The above-presented measurements show that Shawn’s central design goal (simulate the effect caused by a phenomenon, and not the phenomenon itself) indeed leads to a high scalability and performance compared to traditional approaches to simulation. Furthermore, the results show that Shawn’s runtime behavior can be custom-tailored simply by changing configuration parameters. This does not only apply to edge models, but also the other models used in Shawn have a major impact on the performance and resource consumption. Hence, an optimal selection of model implementations must take the properties of the scenario into account.

As a result, developers must carefully select the simulation tool depending on the application area. When detailed simulations of issues such as radio propagation properties or low-layer issues should be considered, Shawn is obviously not the perfect choice. This is where Ns-2 and TOSSIM offer the desired granularity. However, when developing algorithms and high-level protocols for WSNs, this level of detail often limits the expressiveness of simulations and blurs the view on the actual research problem. This is where Shawn provides the required abstractions and performance.
2.2 iSense: A Modular Hardware and Software Platform for Wireless Sensor Networks

iSense is a modular hardware and software platform. It features a number of unique, industry-grade properties. Its modular approach allows creating custom-tailored instances of the hardware as well as the software that comprise exactly the required functionality for a particular application. The small form factor and the available outdoor-capable housings allows for robust and reliable deployments. A low-power design enables a long autonomous operation that – combined with the ability for wireless reprogramming – results in an easy to operate and maintain sensor network. The IEEE 802.15.4 standard-compliant core features a ZigBee-ready radio, hardware encryption and high data rates. The flexible software API with rich a variety of software modules and the use of well-known development tools enables rapid development of market-ready applications.

Hardware overview

In order to fit a wide variety of application demands the iSense hardware platform is made up of a number of modules that can be combined in various ways. Like this, functionality can be easily rearranged, and new features can be added by appending new modules.

Currently, a core module comprising computation and wireless communication, different energy modules, a gateway module for interfacing with computers and an IO-module are available. A number of sensor modules are under development. This module structure is visualized in Figure 22.

The heart of the hardware platform is the core module. It accommodates the wireless micro controller Jennic JN5139, a chip that combines the controller and the wireless communication transceiver in a single housing.

The controller provides 32 bit RISC computation and is running at 16 MHz. It comprises 96kbytes of memory that are shared by program code and data. The advantage of this choice is that memory consumption of program code and data can be traded. Opposite to other controllers where the user is limited to a certain amount of data and code memory, free choices that are only bounded by the sum of both become possible here.

The radio part complies with the IEEE 802.15.4 standard [IEEb]. It achieves a data rate of 250kBits/s and provides hardware AES encryption. With a receive sensitivity of -97dBm and a transmit power tunable between -60dBm and +3dBm it reaches ranges of up to 500m. Apart from this standard version that is equipped with an SMA antenna connector, derivatives with an integrated antenna for especially compact systems or with an additional power amplifier for ranges of up to 2km are available. All three systems are ZigBee-ready.
A common quandary in design is whether or not to use a voltage regulator. It has the advantage that operation with voltages lower than the required one is possible, but the regulator inherently wastes energy. This is especially bad as it also wastes current if the voltage would be high enough and the regulator would not be required. To resolve this problem, we decided to combine the measurement of the supply voltage with the possibility to bypass the regulator by a software switch. Like this, the regulator usage can be omitted when not required but is available when the supply voltage drops.

To enable long, but still synchronous sleep and wakeup cycles, the module is equipped with a high precision clock (error < 20ppm). It features a software switchable LED for debugging purposes.

There is a 34 pin connector on both sides of the module where other modules can be attached to the core module. It can supply up to 500mA to other modules.

The controller can be programmed in various ways. While over-the-air programming (OTAP) is possible and considered to be the standard procedure, the program
can also be transferred via the gateway module (discussed later) or using a special programming adapter that mates with corresponding pads on the module.

The core modules’ sleep current goes as low as 10\(\mu\)A. In full operation the microcontroller uses about 9mA, the radio part 29mA. The module can be powered by a wall mount adapter or a standard battery holder, by one of the power modules or via the USB interface of the gateway module.

Two different power modules are available. The lithium-ion module combines a high capacity rechargeable battery with a charge controller and a battery monitor that tracks the voltage as well as the current flows from and to the battery. The integrated charge controller enables in-system-charging using the core module’s wall mount adapter or the gateway module’s USB power. The coin cell module is intended for particularly compact systems. It holds one CR2477 battery and features a battery monitor for exact battery level information, too.

The gateway module is intended for debugging and interfacing data to other networks. Apart from LEDs, buttons and a user adjustable potentiometer, it features a USB and a RS232 interface. An additional I/O module offers convenient access to the compact bus connector via 2.54mm spaced pins and hence enables fast mock up of adapter boards.

Software overview

The extremely flexible and modular hardware design of the iSense platform requires the same flexibility of the software that drives the individual sensor nodes. The iSense software has been designed with maximal flexibility in mind while allowing for a professional, industry-grade development experience.

One of the fundamental design guidelines is to use state of the art programming methods that are well understood by a large user community. Advanced techniques such as object oriented C++ programming and dynamic memory allocation, that are usually not available in sensor network environments, allow for a rapid and error-avoiding development process. In addition, run-time memory allocation allows for appropriately sized buffers etc. and hence increases memory efficiency. iSense ships with a tiny and lean STL-like implementation that relieves application developers from dealing with recurring and error-prone tasks. It provides implementations for standard containers such as lists, sets and maps. As a result, the development of applications for the iSense platform is completely based on well-known technologies and does not require any proprietary extensions. Hence, the extremely flat learning curve enables a rapid application development that benefits from existing domain expertise of the developers and a plethora of available tools.

Just like its hardware counterpart, the software platform is organized in a set of modules where each of these functional entities provides a highly specialized service to the application. When developing an application, users assemble a subset of the
available modules to a lean operating system that contains exactly the required functionalities. A web-based configuration dialog operated by the coalesenses GmbH allows for an easy, user-friendly selection of these functionalities and subsequently delivers the custom-tailored operating system instance to the user. Figure 23 depicts the overall architecture of the iSense software platform. It is comprised of four distinct building blocks: a hardware abstraction layer on the bottom, operating system functionality and networking support in the middle and the actual user-defined application at the top of the figure.

The hardware abstraction layer (HAL) encapsulates hardware functionality and hides intricate details of the underlying hardware by providing a focused and straightforward application programmer’s interface to the upper layers. Abstractions for interacting with A/D & D/A converters and I/O interfaces (e.g., serial UARTs, I²C and SPI) are available as well as for timers, permanent storage and the wireless interface. A typical usage scenario of the HAL is the integration of the sensor modules described above. These are usually connected to one of the I/O-pins or bus systems and are easily integrated into the iSense software using the HAL functionality. Using this architecture, all modules above the HAL are independent of a particular hardware platform. Application code developed inside this framework is ready-to-run on any platform that provides an implementation of the iSense-API. Currently, it is available in two flavors: one for the iSense hardware platform and one for the simulation framework Shawn [KPB+05, FKFP07]. This allows developers to test their implemented functionality inside a simulation framework before the application is actually deployed on an iSense node thus significantly increasing the development speed.

On top of the hardware abstractions, the iSense framework provides operating system functionalities that ease application development through an event-driven
model. Applications receive call-backs whenever events occur for which the application has registered itself. These events occur either application-driven (e.g., when timers elapse) or hardware-driven (e.g., when input signal of A/D converters change or data is received on one of the I/O-systems). For the application-driven events, the iSense operating system offers two distinct choices. Whenever high timing precision is required, a timing service allows callbacks to be handled uninterruptible and with minimal delay in the interrupt context. Functionality that is not time-critical can register with the tasking service that can be interrupted by the timing service. Finally, the operating system is responsible for conserving the scarce energy resources of a sensor node whenever possible. If desired by the user, the power management infrastructure can put the device into one of the different low power modes.

Besides the functionality that operates strictly local on each single sensor node, one key ingredient of WSNs is wireless communication and consequently a subset of the iSense-modules tackles especially this issue. The HAL already contains convenient abstractions from the details of the wireless interface on top of which the networking support of iSense provides a number of powerful, sophisticated services. It is comprised of routing, time synchronization and over-the-air programming modules. Typical WSN applications require that data is communicated well beyond the communication range of a single sensor node and iSense offers two routing modules that cover a large portion of the design space for sensor network applications. First, a controlled flooding implementation that provides an error-resilient, robust method for conveying data to a set of nodes that is within an n-hop neighborhood of the sending node. Second, a tree-routing module that enables data transfer from the network to one or more sinks. The metric for the link choices is based on packet losses in order to maintain routes with high delivery rates and hence increase network robustness.

It is often vital for WSN applications that the clocks of all nodes in a sensor networks run synchronized, e.g. for data aggregation or sleep/wake-scheduling. This integral feature is integrated as a module in iSense and developers use this functionality and can rely on accurate clocks with a deviation of less than 1ms over 10 hops. Another particularly important module provides the ability to re-program an already deployed sensor network wirelessly. This over-the-air programming (OTAP) provides for a flexible development and operation of sensor networks as wired connections are superfluous and no manual mass-programming is necessary.

Apart from the features of the iSense hardware and software, a comprehensive and accepted development environment is a vital property for successful application development. The iSense software and the development tool-chain are available free of charge and use widely accepted and popular tools such as the Gnu Compiler Collection [Fre84] (GCC) and the Eclipse [Ecl01] development framework. Furthermore, iSense provides iShell, a convenient means to interact with the sensor network. It combines the functionality of a serial terminal, serial and over-the-air programming
of sensor nodes as well as a flexible plug-in system for integrating user-defined functionality such as data analysis or wireless monitoring. iSense and iShell provide an optional (de)-multiplexing service on the serial link. This enables applications to use a number of different, independent data streams, e.g. separating debugging output from different application data streams.
3 Development Environment Setup and Operation

This Section describes the setup and operation of the provided development environment. This covers the simulator Shawn as well as the testbed.

First, Shawn is described in detail, including an installation guide, first steps with the generated executable, documentation of how to extend the source code to write own applications, and the possibility of using JShawn and the GDB.

Afterwards a description of the iSense API, installation instructions for iSense, the conjunction of Shawn and iSense to simulate applications in Shawn that were written with iSense and a guidance for the use of the iSense hardware and programming the sensor nodes are following.

3.1 Shawn Setup

This installation guide describes the required steps for getting Shawn run on Windows and Unix-based systems. There are a few additional tools required, but for each a download line and installation instructions is given.

Windows

Install cmake with cygwin

1. Install cygwin from: http://cygwin.com/. The direct link to download setup.exe is: http://cygwin.com/setup.exe

2. Execute setup.exe. You should use the standard installation path of c:\Program Files\cygwin.

3. Change view to full to have packages sorted in alphabetical order as shown in Figure 24.

![Figure 24: Installing Cygwin.](image-url)
4. Select:
   - `cmake`: A cross platform build manager
   - `gcc-g++`: C++ Compiler
   - `gdb`
   - `make`
   - `unzip`

5. If not already included, add `C:\Program Files\cygwin\bin` to the `PATH` environment variable as follows:
   (a) On desktop: right-click on My Computer and click on properties.
   (b) Click on the Advanced tab.
   (c) Click on the Environment Variables button.
   (d) Highlight the path variable in the Systems Variable section and click edit.
       Add `;C:\Program Files\cygwin\bin` at the end of the path-entries

**Install a Java Runtime Environment**  To use JShawn, you need an installed Java Runtime Environment on your system (minimum required is Java 1.6). You can download it at [http://www.java.com/en/](http://www.java.com/en/).

**Install SVN**  You can download TortoiseSVN here: [http://tortoisesvn.net/downloads](http://tortoisesvn.net/downloads)

**Download Shawn source code**

1. Open your Windows explorer and create a new folder named Shawn somewhere on your disk. Note that the path name must not contain any spaces.

2. If you have installed TortoiseSVN, right-click on your Shawn-folder, select SVN Checkout, and use `https://shawn.svn.sourceforge.net/svnroot/shawn` as the URL for `URL of repository` as shown in Figure 25.

**Generate makefile**  Now we will generate a makefile with the aid of CMake. This makefile is then used to compile Shawn with the GCC.

1. Go to `Shawn/buildfiles` in cygwin shell (e.g.: `cd /cygdrive/C/Programme/Shawn/build-files`)
2. Call `ccmake ../src`

3. Press `c` to create the initial configuration, and wait until it looks as shown in Figure 26.

4. Go down to line `CONFIGURE_APPS` and press `enter` to turn it to `ON`.

5. Again, press `c` to configure. The resulting screen is shown in Figure 27.

6. Now turn all needed modules (starting with `MODULE_APPS`.) to `ON`. Therefore move the cursor to the appropriate line and press `Enter`.

   With respect to `Getting Started`, choosing module apps `*_EXAMPLES`, `*_LOCALIZATION`, `*_READING`, and `*_TOPOLOGY` is a good choice.

   If you do not develop against the iSense-API, **do not** select `*_ISENSE`, because it would result in a linker error.

7. Press `c` again afterwards to update the configuration.

8. Finally, press `g` to generate the makefile that is used for compiling Shawn.
Compile Shawn  Still in Shawn/buildfiles in cygwin shell, call make to start the compilation process. Depending on your CPU, it may take a few minutes. When finished, there is the executable shawn.exe located in the current directory (Shawn/buildfiles).

Import Shawn in Eclipse

1. Download Eclipse from http://www.eclipse.org/downloads/. Either you can choose Eclipse IDE for C/C++ Developers directly, or you should additionally download CDT.

2. Create a new C++-project in Eclipse with the location of your Shawn. Therefore rightclick on the project explorer and select New -> C++ Project. The dialog is shown in Figure 28.

Then, deselect “Use default location”, choose the directory that contains Shawn, and enter a project name. In addition, select “Makefile project” and “Cygwin GCC” as toolchain as in Figure 29.

The next screen should look like Figure 30.
Figure 29: Project options in Eclipse.

Figure 30: Finish new project in Eclipse.
3. Create a new make target for this project in Shawn/buildfiles. For this purpose open the Make Targets-View, go to Shawn -> buildfiles, right-click there and choose Add Make Target. A new window appears that must be completed like shown in Figure 31.

![Create new Make target in Eclipse.](image)

To finish click the Create-Button. With this standard make target, a complex process is started. Dependencies are recalculated, and the CMake build system is checked for any changes. Especially the latter behavior leads to complete re-compilation of the whole code, even when a comment has been added to any CMake configuration file. Therefore it is also possible to use a fast target that only recompiles any changed source file, and rebuilds the binary. The configuration is shown in Figure 32.

Afterwards your Make Targets-View should look like Figure 33.

4. Set in project properties -> C/C++ Build:

   **Build directory:** ${workspace_loc:/Shawn/buildfiles}

   Afterwards it should look like Figure 34.

5. Double click on one of your new created make targets to compile Shawn.

**Linux & Mac OS X**

**Install CMake**

1. If not installed on your system yet, download cmake from: [http://www.cmake.org/HTML/Download.html](http://www.cmake.org/HTML/Download.html) (minimum required is CMake 2.4). Alternatively,
Figure 32: Create target for fast compilation in Eclipse.

Figure 33: Created Make targets in Eclipse.
depending on the used distribution, select the packet from the corresponding packet manager (for example, call `apt-get install cmake` from a shell when using Debian). Either way, make sure that `ccmake`, a curses interface for CMake, is installed (there may be Linux distributions where it is not part of the CMake package).

2. If you build it by hand, unpack the source distributions and follow the instructions in Readme.txt. The steps of installation are:

   i) ./bootstrap
   ii) make
   iii) make install

   from within the unzipped cmake directory.

**Install a Java Runtime Environment**  To use JShawn, you need an installed Java Runtime Environment on your system (minimum required is Java 1.6). You can download it at [http://www.java.com/en/](http://www.java.com/en/).

**Download Shawn**  At first, make sure that Subversion is installed. You can obtain it at [http://subversion.tigris.org/project_packages.html](http://subversion.tigris.org/project_packages.html), or you use the packet manager of the distribution (e.g., type `apt-get install subversion` under Debian systems).
1. Open a terminal and go to the location where your new Shawn directory should be created.

2. Use the svn commandline client to check-out Shawn:

   ```
   svn co https://shawn.svn.sourceforge.net/svnroot/shawn
   ```

**Generate makefile**

1. Go to the folder `shawn/buildfiles` in your terminal.

2. Call `ccmake ..../src`.

3. Press `c` to create the initial configuration, and wait until it looks as shown in Figure 35.

   ![Figure 35: CMake: First view.](image)

4. Go down to line `CONFIGURE_APPS` and press `enter` to turn it to ON.

5. Again, press `c` to configure. The resulting screen is shown in Figure 36.

   ![Figure 36: CMake: Selecting applications.](image)

6. Now turn all needed modules (starting with `MODULE_APPS_`) to ON. Therefore move the cursor to the appropriate line and press Enter.
With respect to *Getting Started*, choosing module apps *EXAMPLES*, *
LOCALIZATION*, *READING*, and *TOPOLOGY* is a good choice.

If you do not develop against the iSense-API, **do not** select *
ISENSE*, because it would result in a linker error.

7. Press c again afterwards to update the configuration.

8. Finally, press g to generate the makefile that is used for compiling Shawn.

**Compile Shawn**  Still in *Shawn/buildfiles* in your shell, call `make` to start the compilation process. Depending on your CPU, it may take a few minutes. When finished, there is the executable *shawn* located in the current directory (*Shawn/buildfiles*).

The latest successfully tested GCC version was 4.3 (Debian 4.3.0-3) 4.3.1 20080401 (prerelease).

**Import Shawn in Eclipse**


2. Create a new C++-project in Eclipse with the location of your Shawn. Therefore rightclick on the project explorer and select New -> C++ Project. The dialog is shown in Figure 37.

![Figure 37: Creating new project in Eclipse.](image)

Then, deselect **Use default location**, choose the directory that contains Shawn, and enter a project name. In addition, select **Makefile project** and **Linux GCC** as toolchain as in Figure 38.

The next screen should look like Figure 39.
Figure 38: Project options in Eclipse.

Figure 39: Finish new project in Eclipse.
3. Create a new make target for this project in Shawn/buildfiles. For this purpose open the Make Targets-View, go to Shawn -> buildfiles, right-click there and choose Add Make Target. A new window appears. Complete it like this:

```
Target Name: shawn
Make Target: all
Build command: make
```

Then, it looks like shown in Figure 40

![Create new Make target in Eclipse.](image)

To finish click the Create-Button. With this standard make target, a complex process is started. Dependencies are recalculated, and the CMake build system is checked for any changes. Especially the latter behavior leads to complete re-compilation of the whole code, even when a comment has been added to any CMake configuration file. Therefore it is also possible to use a `fast` target that only recompiles any changed source file, and rebuilds the binary. The configuration looks as follows:

```
Target Name: shawn_fast
Make Target: all
Build command: make shawnlib/fast shawn/fast
```

The result is shown in Figure 41.
Afterwards your Make Targets-View should look like Figure 42.

4. Set in project properties -> C/C++ Build:

```
Build directory: ${workspace_loc:/Shawn/buildfiles}
```

Figure 41: Create target for fast compilation in Eclipse.

Figure 42: Created Make targets in Eclipse.
Afterwards it should look like Figure 43.

5. Double click on one of your new created make targets to compile Shawn.
3.2 Shawn Operation

When Shawn has been set up successfully, it can be used by either running simulations, or extending the source with own applications. At first, using the generated executable without writing any source code is introduced. After that, how to develop own applications is described. Both, fundamentals and examples are given. Then, the usage of JShawn, a more powerful way to configure and run simulations is presented. At last, a description of using the GDB for debugging is given.

Getting Started

In Getting Started, the usage of the simulator Shawn is introduced. Beginning with a rudimental example that only consists of the leastwise needed commands, a complex and much more meaningful example is developed step by step afterwards. In each step, different features of Shawn are used.

First Simulation

   Step by Step Simulation  After you have successfully compiled Shawn, there is an executable in the buildfiles-directory (usually shawn.exe for Windows users, or just shawn for Unix). When you execute the binary in your preferred shell, the following screen (or a similar one, depending on the applications you have enabled) can be seen in Figure 44.

   Figure 44: Running Shawn the first time.

   For this section, it is essential that you enabled the Examples application when configured shawn using ccmake earlier. The line init_examples(sc) occurs after init_apps (in the picture the output is wrapped) and indicates that this application is enabled.

   We will use the HelloWorld-processor as a first example. As the name implies, it is just a very simple one. After booting, every node sends a message of a special
type. Each node that receives such a message, prints out its own label as well as the label of the sender. If a node does not receive a message for five rounds, it prints out the number of known neighbors (the nodes from whom it has received a message) and the labels (or names) of these neighbors. Then it deactivates itself.

But let us start with the first example. At first, the basic conditions of the simulation environment must be set. Therefore enter the following line in your console to pass it to Shawn:

```
prepare_world edge_model=simple comm_model=disk_graph range=1
```

This command creates a new simulated world and sets its properties such as edge model, communication model, and the communication radius as previously described in Section 2.1. Then, the screen looks like Figure 45.

![Figure 45: Running task prepare_world.](image)

Note that executing a task follows a recurrent process. First, `Simulation: Running task 'name_of_task'` is printed. Then, there follows the output of the task. Finally, the line `Simulation: Task done 'name_of_task'` is printed.

After the simulated world has successfully been created, the world must be filled with life. Therefore we add processors (Section 2.1) of type `HelloWorld` with the following command. Note that this does not create a new world but only adds nodes to the newly created world within the supplied area.

```
rect_world width=25 height=25 count=800 processors=helloworld
```

The command adds 800 processors of type `HelloWorld` in a rectangular part of the world (here of size 25x25), as shown in Figure 46.

At last, the simulation must be started with the following command:

```
simulation max_iterations=10
```

In this example, the parameter `max_iterations` is not required to be given, because the processors set themselves to “inactive” after some rounds (here: after round 6).
But generally, we would advise to set this parameter in each simulation to avoid perpetual runs. However, starting the simulation should look similar to the screens in Figure 47.

First, all nodes print out from whom they received a message. For example, in the first shown line in Figure 47(a), node 341 received a message from node 728. Then, all nodes print the size of their neighborhood, followed by the labels of the neighbors. Node 799, for example, got 6 neighbors with node 141 as the first one, as can be seen in Figure 47(b). The simulation ends with simulation round 6, in contrast to the maximal iteration count of 10. This is caused by the behavior of the nodes which deactivate themselves. Directly after the line that shows the finished simulation round 6, the current number of active, sleeping, and inactive nodes is printed. If the sum of active and sleeping is zero, the simulation finishes.
Using a Configuration File  As you may notice, typing those lines every time you start a simulation is frustrating and exhausting work. Thus, it is also possible to generate a configuration file that contains all the lines you want to pass to Shawn, and then just pass the generated file.

Therefore, create a file named *my_conf* that contains the following lines:

```
# This is a comment that is not processed. Comments *must* be # written in a separate line, and are *not* allowed to be # attached to a line that contains a command for Shawn.
prepare_world edge_model=simple comm_model=disk_graph range=1
rect_world width=25 height=25 count=800 processors=helloworld
simulation max_iterations=10
```

Generally, there are only a few rules when using configuration files. At first, every line that begins with # is comment that is not processed by the parser. Instead, it is continued with the next line. Also, a comment is not allowed to be attached to a valid command for Shawn. Second, a parameter can be made globally available by writing key=value. The key can then be accessed by all following tasks. Finally, a simulation task can be invoked by writing its name. For example, `simulation` invokes the corresponding simulation task *simulation*. Such a task can also be followed by parameter assignment such as the assignment of `max_iterations` after invoking *simulation*. If so, this assignment happens strictly local, and can only be seen by the corresponding task. Subsequent tasks can not access this parameter (if wanted to be global, the assignment must be written in a single line as described before).

A configuration can be passed to Shawn with the parameter `-f`. Hence, start Shawn as follows:

```
./shawn -f my_conf
```

Using multiple Configuration Files  For now, it is not necessary, but if your configuration files get more and more complex, you can also split them into multiple files. Then, you have one master configuration file that you pass to Shawn, and multiple slave files that are included by the master. The master file must contain the following line to include a slave file,

```
# some commands for Shawn ...
include_file slave_file_to_include
# more commands for Shawn ...
```

with *include_file* located exactly at the beginning of the appropriate line in the configuration file. Please note that multiple inclusions are **not** supported (that is, an included file can **not** include other files).
Extending the First Simulation

Playing with Processor Count  The example from *First Simulation* created a world and added 800 nodes within an area of size 25x25 that sent one message at boot time, and then deactivate themselves if they do not receive a message for 5 rounds. To consider a larger scenario we will expand the size of the simulated world, and also put more processors into the world. Hence, change your configuration file as follows:

```
prepare_world edge_model=simple comm_model=disk_graph range=1
rect_world width=50 height=50 count=5000 processors=helloworld
simulation max_iterations=10 connectivity
```

The number of processors increased to 5000, and therefore the size of the world grows to a 50x50 field. In addition, a new task is executed after the simulation finished. The task named *connectivity* is also part of the examples application, and shows the average number of neighbors of the nodes (as well as the minimum and maximum value). However, running this example by also using the *time* command in Linux that shows how long processes ran ends up as shown in Figure 48.

![Figure 48: Running extended configuration with edge model simple.](image)

The execution took approximately 12 seconds, and should have run straight through, but with a noticeable delay after running the task *connectivity*. Now we run the simulation again, but we use an alternative edge model. The configuration looks as follows (note the edge model *list* instead of *simple*):

```
prepare_world edge_model=list comm_model=disk_graph range=1
rect_world width=50 height=50 count=5000 processors=helloworld
simulation max_iterations=10 connectivity
```
The result is shown in Figure 49.

![Figure 49: Running extended configuration with edge model list.](image)

The execution took approximately 7 seconds, and thus was 4 seconds faster than the one with the simple edge model. As you may have noticed, the output (ID 'X' GOT HELLO FROM 'Y') of the simple one was slightly slower than the list one, but there was a delay after starting the connectivity task. Contrary, the task rect_world delayed when using the list model, but everything else executed straight through.

The differences can be explained as follows. The simple edge model should particularly used for small topologies with approximately a few thousand nodes (depending on the applications attached to the nodes). On each transmitted message, all nodes are checked for potential receivers. Then, the task connectivity delayed because for each node in the topology, all nodes were checked for potential neighbors (and counted, if so). In contrast, the list model pre-calculates the neighborhood of each node. Thus, the task rect_world is delayed because for each node in the topology, all other nodes were checked once for being potential neighbors. The neighbors are stored in a list, and used for message transmission (when a node sends a message, the receivers are already known) and neighborhood iteration (just running through the known list). Thus, the execution with the list model was faster than the one using the simple edge model.

**Rerun exactly the same Simulation**  When the task rect_world is used in the beginning of a simulation, the topology is created randomly. Consequently, if you run two simulations one after the other, both runs will probably produce different results. If you want to rerun simulations, you must use the same random seed for both simulations. Shawn therefore provides the simulation task random_seed that can be used for storing and loading the used seed.

Hence, if you run a simulation, add

```plaintext
random_seed action=create filename=file_containing_the_seed
```
as the first line in your configuration file. Then, the used seed can be found in the file 
\textit{file\_containing\_the\_seed}. When you want to rerun a simulation, write

\begin{verbatim}
random_seed action=load filename=file_containing_the_seed
\end{verbatim}

instead. Alternatively, the random seed can also be set directly with

\begin{verbatim}
random_seed action=set seed=123456789
\end{verbatim}

We will use this task in the ongoing examples to point out the purpose of this task.

\textbf{Playing with Transmission Models} So far you have run several simulations, 
from a few hundred nodes to 5000. In addition, you have tried different edge models 
(\textit{simple} and \textit{list}), and you can reproduce simulation results. Now let us have a look 
at the Transmission Model. If not given, the standard transmission model is the 
reliable one. That is, each sent message which can be received by a node is definitely 
received.

In general, it is possible to use multiple transmission models. Each model owns 
therefore the state \textit{chainable}. If set to true, another model can be added. If not, the 
model has to be an end point (that is, it is not allowed to add further models). Message 
sending then looks as follows.

\begin{verbatim}
Message -> ChainTransModel1 -> ChainTransModel2 -> EndTransModel
\end{verbatim}

If a message is sent, it is handed over from model to model, beginning with the first 
in the configuration file. A model is also able to drop messages, so that the message 
does not reach each model at any time.

However, look at the following example:

\begin{verbatim}
random_seed action=create filename=.rseed

prepare_world edge_model=list comm_model=disk_graph \
  transm_model=stats_chain \
  range=1
chain_transm_model name=reliable

rect_world width=50 height=50 count=5000 processors=helloworld
simulation max_iterations=10
connectivity
dump_transmission_stats
\end{verbatim}

First, the seed task has been added that stores the used seed into the file \textit{.rseed}. 
Then, \textit{prepare\_world} got an extra parameter \textit{transm\_model} which is set to \textit{stats\_chain}. 
This model is chainable (as implied by the name) and collects statistics of the overall
sent messages. In the consequence that only information is collected, but no messages are sent, the task `chain_transm_model` in the next line adds the reliable transmission model which transmits each message without delay or loss. The next lines are the same as for the previous simulations, except for the last one. `dump_transmission_stats` is a simulation task that collects the information from `chain_transm_model` and prints it out. The result of running Shawn is shown in Figure 50.

![Figure 50: Running Shawn with StatsTransmissionModel.](image)

After the already known output of the connectivity task there is the result of `dump_transmission_stats`. It shows that 5000 messages were sent (obvious when 5000 nodes send exactly one message at boot time), and that the type of the messages was `HelloWorldMessage`. The output format may differ if you use a different compiler (here: g++ (GCC) 4.2.3), because the return value of `name()` from `typeinfo` is not standardized and depends on the compiler. However, after the simulation finished, the used seed is printed to the terminal (here: 1209547487). Note the number of neighbors of node 4998 (6) as well as the average connectivity (6.1304).

Next, another transmission model is added to the chain. Look at the updated configuration file:

```plaintext
random_seed action=load filename=.rseed

prepare_world edge_model=list comm_model=disk_graph \  
transm_model=stats_chain \  
range=1  
chain_transm_model name=random_drop_chain probability=0.1  
chain_transm_model name=reliable  

rect_world width=50 height=50 count=5000 processors=helloworld  
simulation max_iterations=10
```
dump_transmission_stats

There happened two changes. First, the action of simulation task random_seed changed to load (to use the previous seed of 1209547487). Second, task chain_transm_model is called one more time. It adds transmission model random_drop_chain that drops messages with a given probability (here, set to 10%). Now, the transmission model chain looks as follows.

Message -> StatsChain -> RandomDropChain -> Reliable

If a message is sent, it is first handed to StatsChain that stores count and type of the message. Then, the message is handed over to the RandomDropChain that drops the message with a probability of 0.1. If not dropped, the message is given to the Reliable model that delivers it to the receiver. Have a look at the result shown in Figure 51.

As a result of using the same seed as before, the same scenario as before was simulated. This is indicated by the same average connectivity of 6.1304. But have a look at the neighborhood of node 4998. Now, it shows only 4 neighbors (instead of the 6 before). This is caused by the additional message loss. The last line in the screenshot shows that 481 messages were dropped, and therein contained the two missing HelloWorld messages directed to node 4998.

Visualization The previous example introduced Shawn on the basis of the very simple HelloWorld processor. It only sends messages, and stores the sender when receiving a message. Based on this application, constitutive concepts of Shawn were presented.

Next, we will use a more meaningful application that is also able to produce visual results. After introducing the used application, we create an image of the topology, followed by different enhancements that all show useful features of Shawn.
Executing a more reasonable Simulation  Since you have successfully played with the example application in the previous sections, we will go over to a more meaningful application. Here, it is the localization application that also has an integrated visualization possibility. In general, the application simulates the case when most of the nodes in the network do not know their real position. Indeed, other (fewer) ones called anchors know their real location. The localization application implements different ideas of getting the former ones know their real position based on messages sent out by the anchors (also via multihop).

Make sure that the localization application is set to ON in the CMake configuration (calling ccmake ../src from shawn/buildfiles). Then, generate the following configuration file:

```
random_seed action=create filename=.rseed

prepare_world edge_model=list comm_model=disk_graph \ 
    transm_model=stats_chain \ 
    range=14 
chain_transm_model name=reliable

loc_est_dist=perfect_estimate 
loc_dist_algo=dv_hop 
loc_pos_algo=min_max 
loc_ref_algo=none

rect_world width=100 height=100 count=225 \ 
    processors=localization

localization_anchor_placement anchor_placement=outer_grid \ 
    placed_anchor_cnt=9

simulation max_iterations=50

localization_evaluation
```

First, for reproducibility issues, the used seed is stored in a file named .rseed. Then, the world is prepared by using already known models (including the usage of transmission model chaining), but this time with a greater communication range of 14.

The next lines contain preparation of the localization algorithms. `loc_est_dist` defines how nodes measure the distance between each other. The input must be the name of a Node Distance Estimate model (here, it is the pre-configured perfect_estimate that always returns the correct distance; other choices are described
later). Then, the used algorithms are set. `loc_dist_algo` defines how nodes measure the distance to different anchors. `loc_pos_algo` computes a position from this information. At last, `loc_ref_algo` is responsible for position refinement with the aid of the local neighborhood. There is a more complex but also documented example in `src/apps/localization/randomlocalization.conf` which lists all possible parameters.

Next, a world of size 100x100 is created by putting 225 nodes of type `localization` inside. After that, `localization_anchor_placement` sets 9 nodes to be an `anchor`.

After starting the simulation with a maximum of 50 simulation rounds, `localization_evaluation` prints some information about the results. An example is shown in Figure 52.

First, `localization_evaluation` prints the used algorithms (the ones we have written into the configuration file). Then, it prints some general information. 216 nodes computed successfully their position (remember that we set 9 nodes to be anchors, so 216 + 9 = 225 which is a coverage of 100%).

The next lines contain some additional information about the error rate of the computed positions. E.g., `Average absolute distance from real pos` is approximately 6 which is not a satisfactory result in a 100x100 world and a communication range of 14 (44.6209 percent as can be seen in the figure). In the end of the line, the minimal and maximal error are printed. The best positioned node is located 0.632855 units from its real position, whereas the worst one missed it by 21.7896 units.

The last line gives some information of the connectivity, and prints the average number of neighbors of the nodes (12.6528), as well as the minimum and maximum (2 and 24, respectively).

**Seeing what happens**  Figure 52 showed only textual statistics. On the one hand, very interesting and helpful for the one who evaluates such algorithms. But on the other hand it would be much more exciting if we can see the topology and the false
positioned nodes. Therefore the localization application has an integrated visualization component. If invoked, the task localization_evaluation produces a postscript output of the topology. Change your configuration file as follows:

```
random_seed action=load filename=.rseed
...
localization_evaluation loc_ps_out=topology.ps
```

First, the random seed is reloaded to simulate the same scenario as before. Then, the task localization_evaluation becomes the parameter loc_ps_out that specifies the name of the postscript file that is used for output. After running

```
./shawn -f my_conf
```

there is the file topology.ps in the current folder that contains the pages as shown in Figure 53.

![Figure 53: Postscript-Output from localization application.](image)

The black circles represent the anchors (nodes that know their real position), and the grey nodes represent ones that must compute their real position based on information distributed by the anchors. In the first image you can see additional lines from grey nodes to different positions. Here, the grey nodes are located at their real position, and the lines point to the position where the nodes think they are located. In the second image the nodes are on the positions where they think they are, whereas the third image shows the real topology.

The fact that many nodes point to the same location is based on both the chosen distance algorithm (DvHop) and positioning algorithm (MinMax). We will alter these parameters and compare the results next.

**Comparing different Results** The previously used algorithms led to many nodes that computed the same position. In particular, this is based on the used distance algorithm that only uses hops (times of message forwarding) for distance estimation.
But also the positioning algorithm *MinMax* tends to calculate similar positions from nearly similar data. Hence, let us change both algorithms as follows.

```
loc_dist_algo=sum_dist
loc_pos_algo=lateration
```

Now, the distance algorithm is based on estimation of the real distances. For now, the estimation is done with `perfect_estimate` that returns only correct values. But the next paragraph describes how to introduce ranging errors.

For positioning, *Lateration* is taken that generally computes more accurate values than *MinMax*. Look at the result of the simulation shown in Figure 54.

The most important difference is the average absolute distance from real position which dropped from 6.24693 to 3.7946 (or 44.6209\% to 27.1043\% in percent of communication range). Hence, a noticeable improvement. Figure 55 shows the result from the visualization.
As a result of using the alternative algorithms *SumDist* and *Lateration*, nodes are no longer positioned on equal positions. As already mentioned, this depends particularly on the different distance estimation. Instead of the previously used *DvHop* that computes distances from hops, *SumDist* measures each distance directly. So far, perfect distance estimation is used. That is, if a node measures the distance to another node, it obtains the correct value without any measurement errors. The introduction of such errors is described next.

**Distance Estimation** When a node measures the distance to a neighbor, it can use the available *Node Distance Estimates*. There are three different types available. *Perfect Estimate* returns the real distance without any error. *Absolute Error* uses a uniform random variable for introducing errors, and *Randomized Distance* uses an arbitrary random variable. Because random variables are described later in more detail, the *Absolute Error Distance Estimate* is used first.

The task `create_absolute_error_distance_estimate` for adding such a distance estimate needs at least two parameters. The parameter *name* is used for unique identification, and *error* defines the span of the resulting deviation. There is also the optional parameter *offset* that can be used for a value that is added or subtracted on each estimation. Whenever a node requests a distance, the real distance plus the optional offset plus a random value in the range of \([-\frac{\text{error}}{2}, +\frac{\text{error}}{2}]\) is returned. However, replace the line

```
loc_est_dist=perfect_estimate
```

with

```
create_absolute_error_distance_estimate \
    name=abs_distest error=2.5
loc_est_dist=abs_distest
```

Then, the resulting simulation looks as shown in Figure 56.

The average absolute distance error raised to 6.63457 (or 47.3898% in percent of communication range), and thus even exceeds the results from the first used *DvHop*. Note that the chosen error of 2.5 (-1.25..1.25) is an absolute value, and thus particularly influences nodes that are located close to each other. A relative *Node Distance Estimate* would be a wiser choice, but such objects are introduced later in Section 3.2.

**Topologies** In all previous examples the nodes were placed randomly in a rectangular plane. But Shawn does also allow for more complex topology generation. Therefore the application *topology* must be enabled (check via `cmake .. /src` from
**shawn/build\_files** that **MODULE\_APPS\_TOPOLOGY** is set to **ON**). The topology application provides different tasks for topology generation. One of them generates a network from XML files. As an example, create the file **smiley.xml** with the following content:

```xml
<topology>
  <polygon blocking="0" type="outer">
    <vertex x="0" y="0"/>
    <vertex x="100" y="0"/>
    <vertex x="100" y="100"/>
    <vertex x="0" y="100"/>
  </polygon>
  <polygon blocking="1" type="hole">
    <vertex x="20" y="60"/>
    <vertex x="80" y="60"/>
    <vertex x="80" y="80"/>
    <vertex x="20" y="80"/>
  </polygon>
  <polygon blocking="1" type="hole">
    <vertex x="15" y="15"/>
    <vertex x="15" y="35"/>
    <vertex x="35" y="35"/>
    <vertex x="35" y="15"/>
  </polygon>
  <polygon blocking="1" type="hole">
    <vertex x="65" y="15"/>
    <vertex x="65" y="35"/>
    <vertex x="85" y="35"/>
  </polygon>
</topology>
```
First, the file contains outer tags called `topology` that in turn contain at least one `polygon`. A `polygon` can be either an `outer` one or a `hole`. The former defines the area where nodes are placed. The latter can be used to define holes inside this area in which nodes are not allowed to be located.

When such a XML file has been created it must be loaded by the application. This is done by the simulation task `xml_polygon_topology` which gets the parameters `name` for identifying the polygon by an unique name, and `file` that contains the filename of the XML file (here, it would be `smiley.xml`).

After the polygon has been loaded, nodes must be created in the given area. Therefore the simulation task `populate` must be called. It expects the parameters `topology` (for example, the above generated polygon that is identified by the given `name`), a point generator by `point_gen`, and the amount and type of processors.

Create the following configuration file:

```plaintext
random_seed action=load filename=.rseed

prepare_world edge_model=list comm_model=disk_graph \  
  transm_model=stats_chain \  
  range=4

chain_transm_model name=reliable

loc_est_dist=perfect_estimate
loc_dist_algo=sum_dist
loc_pos_algo=lateration
loc_ref_algo=none

xml_polygon_topology name=xml_top file=smiley.xml
populate topology=xml_top point_gen=uniform_2d count=3000 \  
  processors=localization

localization_anchor_placement anchor_placement=outer_grid \  
  placed_anchor_cnt=25

simulation max_iterations=20

localization_evaluation loc_ps_out=topology.ps
```
Running this configuration in Shawn results in Figure 57.

The corresponding visualization is shown in Figure 58.

![Figure 57: Running first XML topology.](image)

![Figure 58: Postscript-Output of first XML topology.](image)

The nodes have been populated randomly in the specified area, because uniform_2d point generation was chosen. There are also other alternatives for point generation. For example, replace the populate-task in the configuration file with the following line.

```
populate topology=xml_top point_gen=lattice spacing=2 \ 
count=3000 processors=localization
```

The result is visualized in Figure 59.

**Advanced Topics** The previous section introduced many features of Shawn like different transmission models, distance estimation, and the generation of topologies. Nevertheless, there is more functionality available offered by Shawn.
Random Variables  Shawn provides Random Variables for using randomness in applications. By default, there are two different types of probability distribution available: Normal Distribution and Uniform Distribution, but other distributions can be easily added.

The simulation task create_uniform creates a uniform distribution. It takes the arguments lower and upper for defining the range. Whether the given boundaries are included, can be set by lower_incl and upper_incl, respectively. If not given, the task creates a random variable that ranges from 0.0 to 1.0, inclusive 0.0, but exclusive 1.0. Thus,

```
create_uniform name=standard_uniform
```

creates such a variable from. As another example,

```
create_uniform name=next_uni \ 
    lower=-1.8 upper=1.8 \ 
    lower_incl=false upper_incl=true
```

creates a uniform distribution in the interval $(-1.8, 1.8]$.

The other provided possibility is to create a normal distribution with the task create_normal. It takes the arguments stddev for standard deviation, variance, and mean for the base. The appropriate standard values are 1.0, 1.0, and 0.0, respectively. Hence,

```
create_normal name=normal_ex1 mean=1.0 stddev=0.1 
create_normal name=normal_ex2 mean=0.0 variance=0.04
```

are potential calls for creating normal distributions.

Distance Estimation with Random Variables  Previously the distance estimations perfect estimate and absolute error have been used. The former always returns
the correct distance without any error. The latter adds a uniformly distributed value to the returned distance as an error. But there is also the possibility to combine distance estimation errors with own random variables.

The simulation task create_randomized_distance_estimate has been implemented for such cases. Besides the parameters name for unique identification and offset for adding a fixed value on each estimation, it awaits the parameter multiplier. The multiplier is the unique name of an existing random variable. Each time a distance is requested, the real value is multiplied with this random variable. Additionally, the parameter chop_low defines a lower limit of the resulting distance. If lower than the limit, it is either set to the limit directly, or recalculated. The latter happens when resample_chopped is set to true.

For example, a distance estimation with a maximal error of 10%, and all distances greater than zero can be created as follows.

```
create_uniform name=uniform_multiplier lower=.9 upper=1.1
create_randomized_distance_estimate name=uniform_dist \ 
  multiplier=uniform_multiplier \ 
  chop_low=0 resample_chopped=false
```

First, the uniform distribution variable is created with values from 0.9 to 1.1. Whenever a distance is returned, it is first multiplied with this variable.

Alternatively, a similar result but with a normal distribution can be achieved as follows.

```
create_normal name=normal_multiplier mean=1 stddev=0.1
create_randomized_distance_estimate name=normal_dist \ 
  multiplier=normal_multiplier \ 
  chop_low=0 resample_chopped=false
```

**Loading and Saving Worlds**  It is already known that simulations can be rerun by setting the seed with the aid of the simulation task random_seed. But doing so just leads to a complete repetition of the previous simulation. If one wants to run multiple simulations on the same scenario with different results, setting the seed is not an option.

Therefore it is possible to load scenarios from, and store them into XML files. This can be done with the simulation tasks load_world and save_world, respectively. Append the following line to the configuration file that loads smiley.xml and populates nodes in lattice style (should be the last available one):

```
save_world file=test-world-saving.xml
```

The generated file should begin with the following lines (at least similar ones):
The snapshot id is generally set to the current simulation round. In the consequence that the previous simulation set to a maximal iteration value of 20, the value in the XML file is 19 (counting started at 0). The naming of the id can also be changed via the parameter snapshot. There are the magics %r which is default and is replaced by the current simulation round, %n which represents a counter that increases with every save in the actual simulation, and %u for a uuid that is unique for the current run. For example, writing

    save_world file=test-world-saving.xml snapshot=myid:%r_%n_%u

leads to

    [...]  
    <snapshot id="myid:19_0_KMEGUB-A">  
    [...]  

in the XML file. Finally, the generated file is completely rewritten on each run. This can be controlled by the parameter append which is set to false by default. Setting to true allows multiple snapshots per file.

The other way around, each saved snapshot can again be loaded by load_world that requires a filename via file, and can also get an optional name for the snapshot, which is especially useful when multiple snapshots are stored in a file.

**Extending Shawn**

The previous section introduced some basic usage of Shawn by using available and already implemented features. But it is also possible to extend the provided source-code of Shawn with own applications to simulate and evaluate own ideas and concepts. Such extensions follow general development rules that are described next.
First, basic principles of application development are described. Then, the development of a simple example application is shown, followed by the presentation of useful extensions like configuration of a simulation and data extraction.

**Principles of Application Development** The development of applications is based on some basic principles. It is well defined where in the directory hierarchy an application is located, and that each application can be enabled or disabled via CMake. Moreover, it consists of either processors or simulation tasks or both. Finally, also external libraries can be included in the CMake configuration. All of these principles are described next.

**Creating an Application** When adding a new application, the first issue is where to create the new folder. There are two different locations possible: `shawn/src/apps`, and a so called `legacyapps`-folder. The former is directly available after checking out Shawn from svn. It contains well thought-out, reasonable, and thoughtfully implemented applications that follow the general coding guidelines. New applications should only be added with agreement of the project leaders. On the other hand, there is the `legacyapps`-folder that must be created manually. This one can be used for any kind of self-implemented application, because they are not under version control of Shawn. For example, own applications can be held under own version control. After creating this folder, it must be set in the CMake configuration as described later.

After the `legacyapps`-folder has been created, applications can be added. In general they consist of processors and tasks and other code. The files which form an application are organized as a separate software component which is also called a module. The advantage of a module is that it could be in- or excluded from the build process by CMake. Each directory located directly under `legacyapps` is assumed to be a module.

**Creating Legacyapps-Directory** After downloading Shawn, the `src`-directory contains three folders: `apps`, `frontend`, and `sys`. Create a new directory that is called `legacyapps`, copy `apps_init.h` and `apps_init.cpp` to the newly created folder, and rename them to `legacyapps_init.*`. Then, replace every occurrence of `apps` with `legacyapps` in these files.

Change to `shawn/buildfiles` and call `ccmake ../src`. Go to the line named `LEGACYAPPS_PATH`, press Enter, and type the complete, absolute path to the newly created `legacyapps`-folder. Then press Enter again, then c to configure, and at last q to quit. Now, own applications (modules) can be added.

**Creating a Module** To create a new module, create a folder in the `legacyapps`-directory. Here you can create subfolders for your processors and tasks or any fur-
ther code. Then copy an existing module.cmake file (e.g., from any application in shawn/src/apps/) in your folder for your application and change the moduleName to the name of your application/module. Here is a module.cmake you have to create:

```
#=============
# Shawn module configuration for cmake build system
#=============

# Name of this module
set (moduleName SAME_AS_DIRECTORY_NAME )

# Default status (ON/OFF)
set (moduleStatus OFF )

# List of libraries needed by this module, separated
# by white space
set (moduleLibs )
```

The moduleName here is the same as the directory name, but written in upper case characters instead of lower case. You will see the effect while running CMake after pressing c. There is a new menu item like MODULE_LEGACYAPPS_your_module_name (make sure that option CONFIGURE_LEGACYAPPS is set to ON) which could be switched ON or OFF. But how does the compiler know if the code is switched on or off? This is done by the preprocessor directive #ifdef. All your code must be surrounded by #ifdef ENABLE_your_module_name and #endif. Make sure that the directive #include "./buildfiles/legacyapps_enable_cmake.h" is preceding #ifdef ENABLE_your_module_name (in the header-files). Thus, for example, a header starts with

```
#include "legacyapps/your_module_name/your_module_name_init.h"
```

Tasks and processors must be initialized before you can use them in Shawn. This is done by the initialization code. It is organized in the two files your_module_name_init.h and your_module_name_init.cpp. There is only one important method named init_your_module_name. See shawn/src/apps/examples/examples_init.h and examples_init.cpp for an example.

The init_module_name method is called by Shawn when modules are initialized. In the definition file (.cpp file) you have to register your processor-factories and add your tasks to the task_keeper. This is explained later in Section 3.2.
**Writing a Processor**  Processors are useful for distributed implementation of distributed algorithms and can be attached to nodes that in turn are located in the simulated world. It is also possible to attach multiple processors to a node, and thus execute multiple independent algorithms concurrently in a simulation. These algorithms can either run completely independent from each other without even being aware of other processors, or can exchange data over type-safe tags (cp. Section 2.1 or Section 3.2) to solve a global task collectively.

Processors are able to send and receive messages as well as performing a working step in each simulation round. If, for example, there exists a class `MyMessage` that is derived from `shawn::Message`, a processor is able to send such a message via

```cpp
send(new MyMessage);
```

The sent message can then be received by the method `process_message()` that provides a `handle` to the received message. Since a processor can receive different message types, it must be checked if the received one is of the expected type. This is done as follows

```cpp
const MyMessage* msg =
    dynamic_cast<const MyMessage*>(handle.get());
```

If `msg` is not null, it is of the expected type.

Moreover, the virtual method `work()` of a `shawn::Processor` that can be implemented by a derived class, is called automatically in the beginning of each simulation round.

A processor can adopt three different states:

- active,
- sleep, and
- inactive.

An active one is able to receive messages and do the working step in each round, whereas the sleeping one only performs the working step without receiving messages. An inactive processor is out of order and can not get reactivated by itself. Thus, if all processors get inactive during a simulation, the simulation is aborted.

Moreover the methods `boot()` and `special_boot()` can be implemented. `boot()` is called only once (before the first simulation round). `special_boot()` is also called once per processor, and also before the general `boot()` method, but only if the processor is attached to a `special` node. Whether a node is `special` is chosen arbitrarily, but for exactly one node in the network. Hence, the method can be used, for example, for algorithms that require exactly one gateway.

**Writing a Task**  Tasks are useful for centralized algorithms or a small look over the network topology. They are invoked by naming them in the configuration file.
They can be seen as a main-method from where to instantiate certain objects or start certain global actions and algorithms needed in your simulation. They have access to the Simulation Controller and consequently to the whole simulation environment.

To create a new task, it must be derived from `shawn::SimulationTask`, and override methods

- `name()`,
- `description()`, and
- `run()`

where `name()` is used for identifying the task by an unique name, so that it can be invoked from a configuration file, for example. `description()` is used for getting information about a task. Mostly it is a one-liner that roughly describes the purpose of the task. Finally, when a task is invoked, the method `run()` is executed, and provides also access to the simulation controller.

There is an example in `shawn/apps/examples/simulationtask` that can be used as a template for writing an own task.

**Integrating an External Library** If one uses an external library such as Boost, Cairo, CGAL, or the like, it must be integrated into the compilation process. The path of the include files and the path of the library files must be selected. It is possible to add these options to the CMake configuration when calling `ccmake ../src` from buildfiles. An example using the Cairo library looks as follows.

```cmake
OPTION(OPT_ENABLE_CAIRO "Enable Cairo library support" OFF)

if ( OPT_ENABLE_CAIRO )
    set ( DEFAULT_CAIRO_ROOT )
    set ( LIB_PATH_CAIRO CACHE PATH "Path to CAIRO library" )
    set ( INCLUDE_PATH_CAIRO CACHE PATH "Path to CAIRO includes" )
    link_directories( ${LIB_PATH_CAIRO} )
    include_directories ( ${INCLUDE_PATH_CAIRO} )
endif ( OPT_ENABLE_CAIRO )

if ( WIN32 AND NOT CYGWIN )
    if ( OPT_ENABLE_CAIRO )
        if ( LIB_PATH_CAIRO )
            set ( LIB_CAIRO cairo.dll )
        endif ( LIB_PATH_CAIRO )
    endif ( OPT_ENABLE_CAIRO )
else ( WIN32 AND NOT CYGWIN )
```
First, the keyword `OPTION` adds a new option to the CMake configuration, with default turned off. Then, include and library locations are added to the configuration, but only if Cairo is enabled. Then, the libraries given to the linker are added, for Windows as well as other platforms like Linux. At last, the definition for letting the application source code know that Cairo is enabled is added.

Hence, if one sets *Enable Cairo library support* to ON in CMake configuration, and then presses c to configure, the two lines *Path to CAIRO library* and *Path to CAIRO includes* appear as additional options that can be set by the user. On windows systems, for example, the library path is used for finding the file cairo.dll.

**Simple Application**  After learning the fundamentals of application development, we create an own application step by step. It is assumed that the legacyapps folder has been successfully created and integrated into CMake configuration.

The application will be similar to the HelloWorld one in the apps directory. A node sends out exactly one message in the first simulation round. Neighbors receiving such a message add the sender to a private set of immediate neighbors, and print out a debug message. The example will also be extended by parameterized options and the possibility of data extraction later (see Paragraphs 3.2 and 3.2).

**Creating the Files**  First, change to legacyapps and create a folder named `simple_app`. Then copy all files `helloworld_processor*` and `helloworld_message.*` from `src/apps/examples/processor`, as well as `module.cmake` and `examples_init`
from `src/app/examples` to the new directory. Rename `helloworld` and `examples` to `simple_app`. The folder then contains the following files.

```
module.cmake
simple_app_init.h
simple_app_init.cpp
simple_app_message.h
simple_app_message.cpp
simple_app_processor.h
simple_app_processor.cpp
simple_app_processor_factory.h
simple_app_processor_factory.cpp
```

**Adapting the Source** Replace any occurrence of `helloworld` in this files with `simple_app` (and `HelloWorld` with `SimpleApp`, respectively). The `module.cmake` is changed in exactly one line:

```
set (moduleName SIMPLE_APP)
```

The `simple_app_processor.cpp` must be changed in more lines, and is thus completely presented:

```
/**************************************************************************
** This file is part of the network simulator Shawn. **
** Copyright (C) 2004-2007 by the SwarmNet (www.swarmnet.de) project **
** Shawn is free software; you can redistribute it and/or modify it **
** under the terms of the BSD License. Refer to the shawn-licence.txt **
** file in the root of the Shawn source tree for further details. **
**************************************************************************
#include "legacyapps/simple_app/simple_app_processor.h"
#include "sys/simulation/simulation_controller.h"
#include "sys/node.h"
#include <iostream>

namespace simple_app {
  SimpleAppProcessor::
  SimpleAppProcessor()
  {} // ----------------------------------------------------------------------
  SimpleAppProcessor::
  ~SimpleAppProcessor()
  {} // ----------------------------------------------------------------------
  void
  SimpleAppProcessor::
  boot( void )
  throw()
  {} // ----------------------------------------------------------------------
  bool
  SimpleAppProcessor::
  process_message( const ConstMessageHandle& mh )
```
throw()
{
    const SimpleAppMessage* msg =
        dynamic_cast<const SimpleAppMessage*>( mh.get() );

    if( msg != NULL )
    {
        if( owner() != msg->source() )
        {
            neighbors_.insert( &msg->source() );
            INFO( logger(), "Received message from: ",
                  < < msg->source().label() << "" );
        }

        return true;
    }

    return Processor::process_message( mh );
}

void SimpleAppProcessor::
work( void )
{
    throw()
    {
        // send message only in the first simulation round
        if ( simulation_round() == 0 )
        {
            send( new SimpleAppMessage );
        }
    }
}

Also, the simple_app_init.cpp looks different.

Understanding the Source  The generated source contains the minimum for de-
vveloping an own processor that is able to send and receive messages. Let us look at
the files one by one.

First, module.cmake contains the name of the new module to enable configu-
ration by CMake. Change to shawn/buildfiles and run ccmake ../src. Set
MODULE_LEGACY_APPS_SIMPLE_APP to ON.

Then, in the consequence that the processors are able to send and receive own messages, the files simple_app_message.* contain a new message type SimpleAppMessage derived from shawn::Message which in turn provides basic information about sent messages such as source node, size, and timestamp. The presented application uses only the source node of a message, and thus SimpleAppMessage does not get any additional members.

Next, the SimpleAppProcessor has been modified to be a simple version of the HelloWorld application. During boot(), it does nothing. When receiving a message of type SimpleAppMessage, it adds the source of the message to the list of known neighbors and prints a debug message that contains the label of the source. Finally, a message is only sent in the first simulation round. There is an appropriate check in the periodically called work() - method.

The SimpleAppProcessor must also be available for the simulation. Therefore it implements shawn::ProcessorFactory which mainly provides methods name() and create(). The name is used for selecting certain processors when creating nodes. For example, if a configuration file contains the line processors=simple_app, then all available processor factories are searched for the name simple_app. If found, the method create() of the factory is called to create a processor that is assigned to a node (each node gets one).

At last, the implemented processor factory must be added to Shawn’s processor factories so that the processor name simple_app can be found. This happens in file simple_app_init.h where the method register_factory is called which in turn is implemented in SimpleAppProcessorFactory. When enabling a module via CMake configuration, the method init_modulename (here: init_simple_app) is automatically called and must be implemented.

**Compiling and Running** After the source code has been copied and adapted, and the application has been enabled in CMake configuration (setting MODULE_LEGACY_APPS_SIMPLE_APP to ON), Shawn can be compiled to contain the new processor. Therefore type make from shawn/buildfiles directory. When compilation finished, the new processor of type simple_app can be used.

Create a file simpleapp.conf with the following content.

```plaintext
prepare_world edge_model=simple comm_model=disk_graph range=2 rect_world width=10 height=10 count=100 processors=simple_app simulation max_iterations=10 connectivity
```

Calling shawn -f simpleapp.conf results in the execution shown in Figure 60.

The simulation runs exactly as expected. The nodes send a message in the first simulation round (iteration 0), and receive the messages from their neighbors in iter-
Figure 60: Running the Simple Application.

1. Then, the simulation runs up to iteration 9, because we set max_iterations to 10. In contrast to the HelloWorld application, the nodes do not deactivate themselves.

However, we created a very simple processor that only sends out one message in a fixed simulation round, and adds the source of a message to an internal neighbor list on reception, and also prints a debug message. The next step is to parameterize the application, followed by an example for data extraction.

**Configuration of an Application**  It is possible to set parameters in a configuration file, and read them in a processor. Parameters can be integers, strings, boolean values, or floats. Valid assignments are:

- `my_int_value=1`
- `my_string_value=HelloWorld`
- `my_boolean_value=true`
- `my_float_value=3.14`

When processing such a configuration file, Shawn passes these values to the Simulation Environment which in turn acts as a lookup table. Hence, the key `my_string_value` contains the value `HelloWorld`. It can be accessed by the methods

```c
1   required_string_param( "KEY" );
2   optional_string_param( "KEY", DEFAULT_VALUE );
```

When using the method prefixed with the string `required`, and the key is not found, an exception is thrown with a corresponding error message. Generally, the simulation is directly aborted. Alternatively, when using the optional one, a default value can be given that is used if the key is not found within the configuration file.

To get access to the parameters from inside a processor, one must get a reference to the SimulationEnvironment. It can be accessed from the SimulationController.
which in turn is part of the World. The World is available via the owner() of a processor (which in turn is a Node). Hence, change your implementation as follows. Add a new member of type int to the class SimpleAppProcessor.

```cpp
private:
    std::set<const shawn::Node*> neighbors_;  
    int send_round_;  
};
```

Then, adapt the boot() and work() method.

```cpp
void SimpleAppProcessor::
    boot( void )
    throw()
{  
    const shawn::SimulationEnvironment& se =  
        owner().world().simulation_controller().environment();  
    send_round_ = se.optional_int_param( "send_round", 0 );
}
```

```cpp
void SimpleAppProcessor::
    work( void )
    throw()
{  
    // send message only in the first simulation round  
    if ( simulation_round() == send_round_ )  
    {  
        send( new SimpleAppMessage );  
    }
}
```

At last, the used configuration file must contain the new parameter.

```
prepare_world edge_model=simple comm_model=disk_graph range=2  
rect_world width=10 height=10 count=100 processors=simple_app  
send_round=8  
simulation max_iterations=10  
connectivity
```

When booting (method boot() is called when simulation starts; that is, after simulation task simulation is called), the SimpleAppProcessor reads the parameter send_round and writes it to send_round_. If not set, the value is set to the default of 0.

Then, in the work() method of the processor the parameter is checked. If the actual simulation round is equal to the value in the configuration file (or the default value, if not given), a message is sent out. The result is shown in Figure 61.

The messages are sent in simulation round 8, and delivered in round 9, as configured with the parameter send_round_. However, there is one hint left before going on with data extraction from simulation. Note that Shawn related initializations like reading configuration parameters can not be done from the constructor, but only within methods boot(), work(), or process_message(). Within the constructor,
there is no connection to the simulation controller available. A processor is created, and then assigned to the node which in turn provides access to the simulation controller. Thus, do **not** try to access `owner()` in the constructor of a processor.

**Extracting Data from a Simulation** Next thing to show is how to extract data from a simulation. So far, only processors were used that have all their private storage, and each of them is executed once per simulation round. For data collection, however, a simulation task should be used, because then there is only one instance that has access to the whole simulation. When iterating through the nodes to collect the wanted information, the task has particularly access to the `shawn::Node` which in turn owns a collection of processors (at least one). To avoid that the task must access the processor directly, it is possible (for the processor) to add a so called *tag* (see 2.1) to the corresponding node. This tag can in turn be easily read by the task.

**Adding Tags to the Nodes** At first, the processor must add a tag to the node, and write data that can be collected by the task. Therefore the header of `SimpleAppProcessor` is extended by the following lines

```cpp
protected:
void write_int_tag( shawn::Node& node,
    const std::string& tag_name,
    int value ) throw();
```

to declare a new method that writes an integer value to a given tag name that in turn is added to the given node.

The source file must include the integer tag which is a basic one and can be found in `basic_tags.h`:

```cpp
#include "sys/taggings/basic_tags.h"
```

Then, the implementation of the declared method is added
It first checks, if a tag of the given name already exists. In the consequence that we will potentially alter this value, we use `find_tag_w` that returns a writable handle (if there exists any).

If the given name has been found (and thus `tag` is not null), it is checked whether the found tag is an integer one. If so (`intt` is not null), the stored value is updated with the new one. Otherwise, if the tag is not an integer one, we can not write an integer to an unknown tag type, and thus exit without doing anything. Here you see the type-safety of the implemented tagging in Shawn.

If the given tagname could not be found, a new tag is created and attached to the node. The name for the tag as well as the value are given to the constructor. Then, the tag is added to the node.

Next, the method must be called from somewhere. For simplicity reasons, we write the actual number of known neighbors to the tag. Alter the method `process_message` as follows.

```cpp
if( owner() != msg->source() )
{
    neighbors_.insert( &msg->source() );
    INFO( logger(), "Received message from ",
        << msg->source().label() << ",");
}

// write actual number of known neighbors to tag
write_int_tag( owner_w(), "neighbors", neighbors_.size() );
return true;
```

The only new code is the line containing `write_int_tag` (including the corresponding comment), whereas the rest of the shown code is already known. When running the simulation again, each processor will add a tag which contains the actual number of neighbors to the corresponding node.
However, if a node has no neighbors, it would not receive any message and thus would not create any tag named *neighbors*. Therefore we add

```c
// write actual number of known neighbors to tag
write_int_tag(owner_w(), "neighbors", 0);
```

to method *boot*, so that the tag will be initialized in any case.

There is already an example task that is able to iterate through the nodes, and prints all present tags (including value, name of tag, and type). It is part of the examples module in *apps* (and must be enabled when calling *ccmake ../src*). It is called *tagtest*. Hence, when appending *tagtest* to the above generated configuration file, and running Shawn again with this task executed at last results in Figure 62.

![Figure 62: Running task *tagtest* after the simulation.](image)

For each node, it prints the id and the corresponding tags. The appearance of the lines containing tags follows a certain rule:

```
tag_name = value [tag_type]
```

For example, there is one tag attached to node 99 (the one we created). Its name is *neighbors*, its value is 10, and it is of type integer.

**Data Collection with Simulation Task** Since there is one integer tag that contains the actual number of neighbors attached to each node, this data must be collected. The usual way is to implement a simulation task, and start it once after the simulation finished. The task runs through all nodes and reads the appropriate information.

First, the header *collect_simple_app_task.h* must be created that contains the following code.

```c
#define __SHAWN_LEGACYAPPS_COLLECT_SIMPLE_APP_TASK_H
#include "_legacyapps_enable_cmake.h"
ifdef ENABLE_SIMPLE_APP
```
The task CollectSimpleAppTask inherits from shawn::SimulationTask, and thus must implement the methods name(), description(), and run(). As already described earlier, name() defines the command which starts the task when running Shawn. description() can be used to get the purpose of the task. run() is called when the task is started.

There is also the additional method read_int_tag that can read a given integer tag (identified by its name) from the given node, and stores the value. If successful, the method returns true. Otherwise, for example, if the tagname does not exist, it returns false.

The implementation of the task is shown next.
// ----------------------------------------------------------------------
std::string CollectSimpleAppTask::
    name( void )
    const throw()
{
    return "collect_simple_app";
}
// ----------------------------------------------------------------------

std::string CollectSimpleAppTask::
    description( void )
    const throw()
{
    return "Collect information from SimpleAppProcessor";
}
// ----------------------------------------------------------------------

void CollectSimpleAppTask::
    run( shawn::SimulationController& sc )
    throw( std::runtime_error )
{
    require_world( sc );
    int neighbors = 0, count = 0;
    for( shawn::World::const_node_iterator it = sc.world().begin_nodes();
        it != sc.world().end_nodes();
        ++it )
    {
        int tag_value;
        if( read_int_tag( *it, "neighbors", tag_value ) )
        {
            neighbors += tag_value;
            count++;
        }
    }
    if( count > 0 )
    {
        double connectivity = (double)neighbors / (double)count;
        INFO( logger(), "Connectivity: \"\" << connectivity );
    } else
    {
        INFO( logger(), "No tags found." );
    }
// ----------------------------------------------------------------------

bool CollectSimpleAppTask::
    read_int_tag( const shawn::Node& node,
                 const std::string& tag_name,
                 int& value )
    const throw()
{
    shawn::ConstTagHandle tag = node.find_tag( tag_name );
    if( !tag.is_not_null() )
    {
        const shawn::IntegerTag* intt =
            dynamic_cast<const shawn::IntegerTag*>( tag.get() );
        if( !intt )
            return false;
The `read_int_method` gets the node from which it should read the tag, the name of the tag, and a reference to a variable in which it stores the read value. First, it tries to find the tag by its name. If not found, the method returns `false`. Otherwise, it checks whether the found tag is an integer one. If so, it assigns the value and returns `true`. If not, `false` is returned.

The `run()` method works as follows. First, it is checked if the world already exists by the method `require_world`. If not (for example, if the task is called in the first line of the configuration file), an exception is thrown, and the simulation is aborted. Then, it is iterated through all nodes in the world. For each node, it is tried to read the previously written tag (remember method `write_tag` in the processor). If successful, the read value is added to the sum of all neighbors, and a counter is incremented. These two values are used after the loop to calculate the average connectivity in the network. If not possible (no tags could be read), an appropriate message is printed.

The next step is to add the task to Shawn, so that it can be found if called in a configuration file. That means, the task must be added to the simulation task keeper that is part of the `SimulationController`. The usual location for doing so is the `init` method. Hence, `simple_app_init.cpp` is altered as follows.

```cpp
#include "legacyapps/simple_app/simple_app_processor_factory.h"
#include "legacyapps/simple_app/collect_simple_app_task.h"
#include "sys/simulation/simulation_controller.h"
#include "sys/simulation/simulation_task_keeper.h"

extern "C" void init_simple_app( shawn::SimulationController& sc )
{
    simple_app::SimpleAppProcessorFactory::register_factory( sc );
    sc.simulation_task_keeper_w().add( new simple_app::CollectSimpleAppTask );
}
```

In the consequence that a new file has been added (the new task), that file must be made public to CMake. So either rerun `ccmake`, or alternatively edit (add and remove a space) and save `module.cmake`. Then recompile.

Before running the simulation again, change the configuration file as follows.

```
prepare_world edge_model=simple comm_model=disk_graph range=2
rect_world width=10 height=10 count=100 processors=simple_app
send_round=8
simulation max_iterations=10
collect_simple_app
connectivity
```
In the end of the task there were two tasks added. The newly implemented `collect_simple_app`, and connectivity to compare the results. An example is shown in Figure 63.

![Figure 63: Running new task collect_simple_app.](image)

Both tasks show an average connectivity of 10.64. The former computed the value by exchanging messages (and thus the transmission model). The latter used the internal communication model for acquiring the information.

**JShawn**

As already described and used before, Shawn provides the use of configuration files. Parsing such files follows very simple rules. Either you can set a parameter with `key=value`, or you can invoke a task by writing its name. When the former is used, `key` is set to `value`, and can be looked up in the simulation environment. But there is no possibility to use loops, if clauses, or local variables within the configuration file. Therefore JShawn has been developed. It requires at least Java 1.6, and allows for writing simple *scripts*.

**Syntax**  
For introducing scripting functionality, JShawn uses the BeanShell \(^1\). It allows the execution of the full Java syntax, Java code fragments, as well as loosely typed Java and additional scripting conveniences. Also, it enables transparent access to all Java objects and APIs.

The most important issue is how to access Shawn from a JShawn configuration file. There are two commands available:

```
1  shawn.setGlobalVariable( "variable", "value" );
2  shawn.runCommand( "task-name", "settings_of_required" 
3       "and_optional_parameters" 
4       "of_the_named_task" );
```

\(^1\)http://www.beanshell.org
The former is used to set global variables, such as key=value in the well-known configuration files. The latter invokes the given task, and adds the possibility for setting local variables that can only be seen by the task (and thus are not globally available).

Next, printing debug messages is done the same way as in Java with `System.out.println`:

```java
System.out.println( "my␣debug␣message" );
System.out.println( "i␣=␣" + i );
```

Local (script-wide) variable declaration works as follows.

```java
double my_double = 0.123;
int my_int = 123;
```

Finally, also loops and if-clauses can be used easily:

```java
for ( int i = 0; i < 10; i++ ) ...
if ( foo ) {
    // do something
}
else {
    // do something else
}
```

**Simple Example**  In Section 3.2, we used the following configuration file as a first simulation:

```
prepare_world edge_model=simple comm_model=disk_graph range=1
rect_world width=25 height=25 count=800 processors=helloworld
simulation max_iterations=10
```

To convert the file into a JShawn configuration file, create a file named `simple-jshawn.bsh`, and add the following lines:

```java
shawn.runCommand( "prepare_world", "edge_model=simple␣" +
    "comm_model=disk_graph␣" +
    "range=1" );
shawn.runCommand( "rect_world", "width=25␣height=25␣" +
    "count=800␣" +
    "processors=helloworld" );
shawn.runCommand( "simulation", "max_iterations=10" );
```

Then, the simulation can be started via the commandline when running from the main Shawn directory:

```
java -jar utils/jshawn-allinone.jar
   -s buildfiles/shawn
   -b simple-jshawn.bsh
```

The result can be seen in Figure 64.

When starting JShawn as shown in Figure 64(a), a new instance is created that processes the given script. JShawn executes Shawn, reads the output, and passes input lines to the executable. At first, it begins with own debug output describing
that a new instance is created. Then, it redirects output from Shawn. For example, the line beginning with `init_apps` is read from Shawn, and directly printed to standard out. The line beginning with `/ NOW READING...` (cp. Figure 45) on the other hand is completely suppressed. Much the same happens with the output printed when tasks start and finish. Output for starting tasks is suppressed, whereas a finished task is shown by an own debug message. However, after the simulation finished, there follows concluding debugging output as shown in Figure 64(b).

**Complex Example** The previous example was very simple, and was a direct conversion from a Shawn configuration file. It used only the possibility of starting tasks in Shawn. Next, we will use a more complex configuration file that also uses variables, loops, if clauses, and debug output.

```java
shawn.setGlobalVariable("name", "xml_top");
shawn.runCommand("xml_polygon_topology","file=smiley.xml");

boolean lattice = false;

for ( int i = 1; i <= 10; i++ )
{
    System.out.println( "i = " + i);
    shawn.setGlobalVariable("processors", "helloworld");
    shawn.runCommand("prepare_world", "edge_model=simple" +
                     "comm_model=disk_graph" +
                     "range=10");
    if ( lattice )
    {
        shawn.runCommand( "populate", "topology=xml_top.." +
                           "point_gen=lattice.." +
                           "spacing=" + i + ",count=200");
    }
    else
    {
        shawn.runCommand( "populate", "topology=xml_top.." +
                           "point_gen=uniform_2d.count=200");
    }
```
The example uses the already previously described XML topologies. Again, it uses the `smiley.xml` that produces a topology like the one shown in Figure 58. It is also possible to place the nodes on a lattice, but this is done later on. For now, we set the internal script-wide variable to `false`. Then, a for loop iterates ten times through separate simulation processes. At first, the current simulation number is printed. Then, the type of used processors is set to `HelloWorld`, and the world is prepared by using a simple edge model, and a unit disk graph communication model with a transmission range of 10. If the variable `lattice` is set to `true`, 200 nodes are placed on a lattice. Otherwise, 200 nodes are randomly populated in the world. At last, the simulation is started with a maximal iteration count of 50, the actual topology is printed as postscript output, and the world is saved to a XML file.

So far, the task `polygon_topology_postscript` has not been presented. It visualizes three parts of the topology. First, the topology borders from both inner and outer polygons are printed. Second, connections between nodes that are in communication range are represented by lines. Third, nodes are printed as circles.

The for loop runs 10 times, and thus there are 10 postscript files named `topoX.ps` after running the script. The nodes were populated randomly, so that there are 10 files with arbitrary topologies, named `topo1.ps` to `topo10.ps`. Two examples are shown in Figure 65.

![Figure 65: Running Topology script in JShawn with random topologies.](image)
Next, we will change the script to populate the nodes on a lattice. Therefore we write

```java
| boolean lattice = true;
```

in line 4 of the script. The for loop runs from 1 to 10, and sets the spacing of the lattice dynamically to the current iteration. Figure 66 shows two examples, one with a spacing of 5, the other one with a spacing of 9.

![Lattice examples](image)

(a) Lattice with spacing of 5  
(b) Lattice with spacing of 9

Figure 66: Running *Topology* script in JShawn with random topologies.

### Debugging with GDB

If an error occurs somewhere in your application, you typically use a debugger that supports you in finding the bug. When using the GCC, the GNU Project debugger (GDB) is the naturally used tool. Make sure that the GDB is installed on your system. Using Windows, see Section 3.1 how to set up Cygwin correctly. For Unix users, use the packet manager of your distribution, or go to [http://sourceware.org/gdb/](http://sourceware.org/gdb/) alternatively and follow the installation instructions.

Running Shawn in GDB requires some specific compilation settings, and then a complete recompilation of your code. To prepare Shawn, execute the following steps:

- Change to `shawn/buildfiles` using your preferred Shell.
- Type `ccmake ..../src`
- Change `CMAKE_BUILD_TYPE` to either `RelWithDebInfo` or `Debug` as shown in Figure 67.
- Press `c` to take the actual configuration.
- Press `g` to generate makefiles.
• Finally, type `make` to recompile Shawn, so that it can be used in GDB.

As you may notice, Shawn executable is considerably bigger than the release version. However, now you can start debugging your application. Make sure that the simple configuration file from Section 3.2 is available in the current directory. Located in `shawn/buildfiles`, type

```
gdb shawn
```

to start GDB (alternatively, type `gdb shawn.exe` for Windows users in Cygwin shell). Then the screen should look like Figure 68.

```
Figure 67: Preparing Shawn for being used by GDB.
```

```
Figure 68: Starting GDB.
```

Now that GDB has been started with the Shawn binary as argument, Shawn must be executed. There are two possibilities of doing so. Either Shawn can be run by typing `r` or `run`, or a configuration file can be given additionally by typing `r -f file.conf`. The former allows for passing commands line by line, whereas the latter uses the given configuration file for execution. However, both options are shown in Figure 69.
3.3 Application development with iSense

This Section deals with the development of applications with the iSense API. It shows how to build up an executable iSense application, how to simulate it with Shawn and run it on the iSense hardware.

Shawn simulations using the iSense API

In the following it is described how to run applications that are implemented with the iSense API as simulations with iSense/Shawn – the internal Shawn-version that is provided with iSense. Installation instructions are given, followed by an illustration of the iSense API, a description about how to implement applications with the iSense API and simulate them with iSense/Shawn afterwards.

How to install iSense

The precompiled version of iSense that you can obtain on http://www.coalesenses.com/index.php?page=webcompile consists of precompiled solutions for the different Jennic platforms and the simulator Shawn.

In the following there are instructions given on how to install iSense with webcompile on Windows.

1. Installing cygwin

(a) Running Shawn via r

(b) Running Shawn via r -f simpleapp.conf

Figure 69: Starting Shawn in GDB.

If something abnormal happens, for example if the program crashes, one can call bt for backtracing what happened just before the crash. Also, breakpoints can be set via break before starting Shawn. When finished, one can call q for exiting GDB.

However, it is highly recommended to consult the GDB documentation available under http://sourceware.org/gdb/download/onlinedocs/gdb.html, or at least typing help in the GDB command line interface.
(a) Install cygwin from: http://cygwin.com/. The direct link to download setup.exe is: http://cygwin.com/setup.exe

(b) Execute setup.exe. You should use the standard installation path of C:\Program Files\cygwin.

(c) Change view to full to have packages sorted in alphabetical order as shown in Figure 70.

![Figure 70: Installing Cygwin.](image)

(d) Select:
   - `cmake: A cross platform build manager`
   - `gcc-g++: C++ Compiler`
   - `gdb`
   - `make`
   - `unzip`

(e) If not already included, add C:\Program Files\cygwin\bin to the PATH environment variable as follows:
   
i. On desktop: right-click on My Computer and click on properties.
   
   ii. Click on the Advanced tab.
      
   iii. Click on the Environment Variables button.
      
   iv. Highlight the path variable in the Systems Variable section and click edit.
       
       Add ;C:\Program Files\cygwin\bin at the end of the path-entries.

2. Install Java

   To use JShawn (a tool to use the Java language for parameterizing and running Shawn and iSense/Shawn simulations), Java must be available on your system. The minimum requirement is Java 1.6. You can download Java at http://www.java.com/en/.

3. Install the ba-elf-cross-compiler
To develop, compile and link software for the iSense hardware platform, a cross-compiling version of the GCC compiler is required. To install this compiler, perform the following steps:

(a) Go to http://www.beyondsemi.com/page/products/softwaretools and download the BA elf binaries for Cygwin (you must register to do so).

(b) Extract the archive to the cygwin directory. Note that you have to unzip the archive in a cygwin console! Using any other Windows tools to unzip will supposably cause errors within the further steps.

(c) Go to C:\Program Files\cygwin\opt. Create a symbolic link ba-elf in opt to the created directory by typing `ln -s ba-elf-r11976.i686-cygwin ba-elf`

(d) Add C:\Program Files\cygwin\opt\ba-elf\bin (or the corresponding path on your system) to the PATH environment variable as follows:
   i. On desktop: right-click on My Computer and click on properties.
   ii. Click on the Advanced tab.
   iii. Click on the Environment Variables button.
   iv. Highlight the path variable in the Systems Variable section and click edit. Add 
      ;C:\Program Files\cygwin\opt\ba-elf\bin
      at the end of the path-entries.

4. Download iSsense

(a) Open web browser and go to http://www.coalesenses.com/index.php?page=webcompile
   where you will find the iSense configuration interface (cp. Figure 71).

(b) Define platforms and parameters to your needs in the GUI. By default, a configuration that should be helpful for most users is selected. If you select an option that requires other options, and those other options are not selected yet, a dialog pops up and asks whether those choices should be done automatically (cp. Figure 72). Choose "Ok". After making all your choices, click on ”Build iSense“ and wait for the compilation process to complete (cp. Figure 73). This can take some minutes.

(c) After the compilation completes, click on ”Download“ to access the compiled firmware zip file isense.zip (cp. Figure 74).

(d) Unzip the archive isense.zip into a new folder named iSense. Take notice of the case sensitivity here. Also note that you have to unzip the archive in the cygwin console! Therefore you should have installed the unzip command in the cygwin setup some steps ago. Using any other Windows tools to unzip will supposably cause errors within the further steps.
Figure 71: Webcompile: iSense configuration interface

Figure 72: Webcompile: Message for resolving dependencies
Figure 73: Webcompile: Compilation process

Figure 74: Webcompile: Download iSense
(e) Obtain the isense-sdk.zip file from www.coalesenses.com/download/isense_sdk.zip that contains all the non-customizable components, and extract it to a location of your choice. Note that you have to unzip the archive in the cygwin console here, too. Using any other Windows tools to unzip will supposably cause errors within the further steps. The archive contains an empty directory ”iSense”.

(f) Replace the empty directory iSense in the unzipped isense_sdk-folder by your own iSense-folder that you created with webcompile some steps ago.

5. Using Eclipse for C / C++ development

If you have not already installed Eclipse on your system install Eclipse IDE for C/C++ Developers. An installation instruction follows here:

(a) Download the Eclipse ZIP-file (Eclipse IDE for C/C++ Developers) from http://www.eclipse.org/downloads/

(b) Extract the content e.g. to C:\Program Files\n
(c) Launch Eclipse

If Eclipse is already installed on your system just download and install CDT for Eclipse. Have a look at the following installation instructions therefor.

(a) Download CDT from http://www.eclipse.org/cdt/downloads.php and unzip it

(b) Choose in Eclipse: Help -> Software Updates -> Find and Install

(c) -> Search for new features to install

(d) -> New Local Site

(e) Choose the subdirectory eclipse in the directory where you stored CDT and install it

6. Import iSenseDemoApplication into Eclipse

(a) Open Eclipse

(b) If not chosen, choose Window -> Open Perspective -> Other -> C++

(c) Choose File -> Import

(d) Then go to General -> Existing Projects into Workspace and choose the root directory of the isense_sdk folder

(e) Check the box iSenseDemoApplication and click on Finish. Now the project iSenseDemoApplication should appear in your Eclipse Project Explorer. Open iSenseDemoApp.cpp to edit the application.
(f) To compile the application for your device go to the Make Targets-View and double-click on the make target appropriate to your device. If that does not work, check whether you have added C:\Program Files\cygwin\bin to the PATH environment variable (see "Installing cygwin" above).

7. Import own applications into Eclipse

(a) Duplicate the folder iSenseDemoApplication and rename it as you like to (e.g. TogglingDemoApplication)
(b) Open the .project- file and search for the line
   
   \texttt{\textless \text{name}iSenseDemoApplication\textgreater
   \text{\textless\textgreater}}

   at the beginning of the file. Replace it with the name that you have chosen for your new application folder a step ago, e.g.
   
   \texttt{\textless \text{name}TogglingDemoApplication\textgreater
   \text{\textless\textgreater}}

(c) Open Eclipse
(d) If not chosen, choose Window -> Open Perspective -> Other -> C++
(e) Choose File -> Import
(f) Then go to General -> Existing Projects into Workspace and choose the root directory of the isense-sdk folder
(g) Check the box TogglingDemoApplication (or the box with the name you have chosen for your own application) and click on Finish. Now the new project should appear in your Eclipse Project Explorer. Open iSenseDemo.cpp to edit the application.
(h) To compile the application for your device go to the Make Targets - View and double-click on the make target appropriate to your device.

8. Subscribing to the iSense Mailing list

There is a mailing list for iSense users: isense@coalesenses.com To subscribe, send an email with the subject subscribe isense and content subscribe isense to majordomo@coalesenses.com

9. How to install doxygen

To generate the API documentation for iSense, Doxygen must be installed. To install this tool, perform the following steps:

(a) Download Graphviz from \url{http://www.graphviz.org/} and install it to C:\Program Files\Graphviz
(b) Download Doxygen from
ftp://ftp.stack.nl/pub/users/dimitri/doxygen-1.5.6-setup.exe or
http://ftp.stack.nl/pub/users/dimitri/doxygen-1.5.6-setup.exe
and install it

c) Start the Doxygen GUI (Doxywizard) and load the configuration file doxy-conff.new from your isense_sdk\iSense folder

d) Now choose Expert, go to the Dot tab, and adapt the DOT_PATH to the bin directory of your GraphViz installation (C:\Program Files\Graphviz\bin)

(e) Save the configuration by clicking on the Save... button unter Step 2

(f) To generate documentation, click on Start, and wait for the process to finish

(g) Now you can find the Doxygen-documentation of the iSense API in your iSense folder under doc\html

**iSense API**  This Section describes the structure and the modularity of iSense in the first Subsection ("iSense modular firmware and simulation environment"). The second Subsection ("iSense application development") contains guidances for

- Adapting the iSense build environment
- Getting started with iSense
- An Example Application: iSenseDemoApplication
- Extending the iSenseDemoApplication: TogglingDemoApplication

**iSense modular firmware and simulation environment**  The flexible and modular hardware design of the iSense platform requires the same flexibility of the software that drives the individual sensor nodes. The iSense software has been designed with maximal flexibility in mind while allowing for a professional, industry-grade development experience. One of the fundamental design guidelines is to use state of the art programming methods that are well understood by a large user community. Advanced techniques that are used are object oriented C++ programming and dynamic memory allocation, that are usually not available in sensor network environments. This allows for appropriately sized buffers etc. and hence increases memory efficiency.

All firmware functionality is structured into modules that can be individually configured to be part of the firmware or not. Like this, lean but comprehensive software that contains exactly the features required for the target application can be developed.

The firmware is divided into three parts (compare Figure 23):
• The Hardware Abstraction Layer (HAL) encapsulates hardware-specific drivers for the radio, A/D & D/A converters and I/O interfaces
• Operating system functionality such as power management, tasking or timing
• Protocol suites including routing, transport, time synchronization, location awareness or over-the-air programming

The HAL encapsulates hardware functionality and hides intricate details of the underlying hardware by providing a focused and straightforward application programmer’s interface to upper layers. Abstractions for interacting with A/D & D/A converters and I/O interfaces (e.g., serial UARTs, I²C and SPI) are available as well as for timers, permanent storage and the wireless interface.

Using this architecture, all modules above the HAL are independent of a particular hardware platform. Application code developed using this framework is ready-to-run on any platform that provides an implementation of the iSense-API. Currently, the iSense-API is available in two flavors: one for the iSense hardware platform and one for the simulation framework Shawn. This allows developers to test their implemented functionality inside a simulation framework before the application is actually deployed on iSense nodes, thus significantly increasing the development speed.

On top of the hardware abstractions, the iSense framework provides operating system functionalities that ease application development through an event-driven model. Applications receive call-backs whenever events occur for which the application has registered itself. These events occur either application-driven (e.g., when timers elapse) or hardware-driven (e.g., when input signals of A/D converters change or data is received on one of the I/O-systems). For the application-driven events, the iSense operating system offers two distinct choices. Whenever high timing precision is required, a timing service allows callbacks to be handled uninterruptible and with minimal delay in the interrupt context. Functionality that is not time-critical can register with the tasking service that can be interrupted by the timing service. Finally, the operating system is responsible for conserving the scarce energy resources of a sensor node whenever possible. If desired by the user, the power management infrastructure can put the device into one of different low power modes.

Besides the functionality that operates strictly local on each single sensor node, one key ingredient of WSNs is wireless communication and consequently a subset of the iSense-modules tackles especially this issue. The HAL already contains convenient abstractions from the details of the wireless interface on top of which the networking support of iSense provides a number of powerful, sophisticated services.

Typical WSN applications require that data is communicated well beyond the communication range of a single sensor node and iSense offers a number of routing modules that cover a large portion of the design space for sensor network applications. First, a flooding implementation that provides an error-resilient, robust method for conveying data to a set of nodes that is within a n-hop neighborhood of the sending
node. Second, a tree-routing module that enables data transfer from the network to one or more sinks. The metric for the link choices is based on packet losses in order to maintain routes with high delivery rates and hence increase network robustness. This metric is also used by two other routing schemes that provide a one-to-one-semantic, creating either unidirectional or bidirectional paths through the network. In addition, a number of traditional ad-hoc routing protocols such as DSDV [PER94] or LMR [COR95] have been integrated into iSense.

Besides routing, a highly accurate time-synchronization scheme that provides an accuracy of less than 1ms is part of iSense as well as the well-known localization protocols described in [LAN03]. They are augmented by reliable multi-hop transport and over-the-air programming modules.

iSense ships with a tiny and lean STL-like implementation that relieves application developers from dealing with recurring and error-prone tasks. It provides implementations for standard containers such as lists, sets and maps. As a result, the development of applications for the iSense platform is completely based on familiar technologies and does not require proprietary extensions. Hence, the extremely flat learning curve enables a rapid application development that benefits from existing domain expertise of the developers and a plethora of available tools.

Apart from the features of the iSense hardware and software, a comprehensive and accepted development environment is a vital property for successful application development. The iSense software and the development tool-chain are available free of charge, and use widely accepted tools such as the Gnu Compiler Collection (GCC) and the Eclipse development framework. Furthermore, iSense provides iShell, a convenient means to interact with the sensor network. It combines the functionality of a serial terminal, serial and over-the-air programming of sensor nodes as well as a flexible plug-in system for integrating user-defined functionality such as data analysis or wireless monitoring. iSense and iShell provide an optional (de)-multiplexing service on the serial link. This enables applications to use a number of different, independent data streams, e.g., separating debugging output from application data streams.

**iSense application development** This section describes how to develop applications with iSense. It introduces the basic concepts of the iSense API and also exemplifies those with source code of complete applications.

1. Adapting the iSense Build Environment

   You are able to fundamentally adapt your iSense build environment to your needs by defining platforms and parameters in the GUI at [http://www.coalesenses.com/index.php?page=webcompile](http://www.coalesenses.com/index.php?page=webcompile) and building a new iSense for your system there with `webcompile`.

2. Getting started with iSense
In the following the basic concepts of iSense are introduced.

An iSense application substantially consists of

- a `boot` -
- an `execute` -
- a `receive` -
- a `timeout` - and
- an `application_factory`

- method.

The `application_factory`-method instantiates a new application. After starting the application, the `boot`-method is called. Normally you would add tasks or timeouts, set the state of the sleep-mode or write debug-outputs in here. Generally it is to take heed of being quick in the `timeout`-method because it is always called in the interrupt context. You can use tasks instead if you need more time.

The main-work of the application takes place within the `execute`-method. If an application registers itself as a task, the OS will call the application’s `execute`-method, when the associated task is due. Within the `receive`-method you can handle incoming messages.

So before you begin to develop your own application also the following points may be a matter of particular interest, but please note that the following annotations are explained in the iSense API documentation in detailed fashion. So have a look at it for further information.

(a) The application’s `boot`-method

Any iSense application must inherit from `isense::Application`. On boot of the iSense-OS, the application is instantiated by invoking a method with the following signature:

```
isense::Application* application_factory(isense::Os &os).
```

Implement this method in your cpp-file and return a new instance of your application. Please note that this method may be called multiple times when used in conjunction with the iSense/Shawn-environment since in the simulation, a multitude of nodes require their own Application instance. A direct consequence of this concept is that your application must be aware that multiple instances of your class may coexist in the same simulator instance. Therefore, variables must not be static outside of your class’ scope.

After the single or multiple instances of your application have been instantiated, the application’s `boot`-method is called. This method must not perform a blocking operation such as a `while(1) ...;` loop, but must return after initialization is done. Any further action performed by your application must
therefore be event-driven by external events such as timers, packet reception, I/O, etc.

(b) Tasks and Timeouts

To get notified at certain points in time, iSense offers Tasks and Timeouts. The main difference between both is the context in which they are invoked. Timeouts are called in the interrupt context, so the timeout-method should only run for an extremely short period of time. Tasks are invoked in the normal context so lengthy pieces of work (basically everything that requires more than a few lines of code) should be done here. Tasks are organized in a fifo-queue and are executed one after the other. If the queue is empty the device enters a sleep mode (only if the application has enabled sleeping). Another difference between Tasks and Timeouts is that scheduled Timeouts can be removed while Tasks can not be removed.

(c) Transmitting and Receiving packets

• Transmitting

To transmit packets, you must use an instance iSense::Radio class which is obtained from the isense::Os::radio(). The send-method of the isense::Radio requires the destination address, the length of the buffer to be transmitted, the buffer, options and optionally a pointer to isense::Sender instance. For instance, a typical application would look like this:

```
os_.radio().send(ISENSE_RADIO_BROADCAST_ADDR, len, outbuffer,
                    Radio::ISENSE_RADIO_HEADER_OPTION_NONE, this);
```

The destination address is a 16-bit value indicating the intended receiver of the packet. If you specify ISENSE_RADIO_BROADCAST_ADDR, this packet is received by any other device in the broadcasting range. Amongst others, the options tell the radio to perform acknowledged or unacknowledged transmission. To use the default options, specify Radio::ISENSE_RADIO_HEADER_OPTION_NONE.

The sender parameter may be NULL or point to a class implementing the isense::Sender interface. If this parameter is not NULL, the radio will notify the sender about success or failure. If you send without requesting an ack, the confirm-method of a sender is called in the interrupt context as soon as the packet has been sent by the radio. If you have requested an ack, it is called as soon as the acknowledgement arrived or after n failed tries.

• Receiving

To receive packets, the class that should receive packets must implement the isense::Receiver interface. Second, you must register this receiver
with the \textit{isense::Dispatcher} in order to actually receive packets. This could be done in the application’s \textit{boot}-method as follows:

\begin{verbatim}
    os_.dispatcher().register_receiver(this);
\end{verbatim}

It is the task of the \textit{isense::Dispatcher} to dispatch incoming messages to registered receivers. A standard \textit{isense::Receiver} receives packets addressed to its device’s MAC address and the broadcast-address. Its \textit{receive}-method is called in the normal context using the same mechanism as Tasks. If desired, a receiver must check whether a message has been sent to the broadcast-address or to the current device and, e.g., drop packets to the broadcast address if the receiver is not interested in receiving broadcasted messages.

Another option is to register a so-called \textit{fast receiver}. A fast receiver obtains the same packets, but is called in the interrupt context. There can only be one fast receiver. To register as a fast receiver, do the following:

\begin{verbatim}
    os_.dispatcher().register_fast_receiver(this);
\end{verbatim}

A \textit{promiscuous receiver} obtains all packets the device hears, even though the destination address differs from the own one.

\begin{verbatim}
    os_.dispatcher().register_promiscuous_receiver(this);
\end{verbatim}

(d) \textbf{SleepHandler}

An application can extend \textit{SleepHandler}. If it registers itself at the \textit{Os}, it will be called before going to sleep and after wakeup. Be quick in both methods \textit{standby} and \textit{wakeup}, because otherwise the internal time or wakeup time may get wrong.

(e) \textbf{Peripherals}

If you need to interact with the peripherals of an iSense device, references to these system components like GPIO, I\textsuperscript{2}C, UART and ADC may be obtained from the \textit{isense::Os}.

3. Example application: \textit{iSenseDemoApplication}

The example application (\textit{iSenseDemoApplication}, Listing 1) demonstrates the basic steps required for most iSense applications.

Within the \textit{boot}-method there is a debug output \textit{App::boot}. Because this method is called as soon as the application is started, you should see this output in the debug-window. Thus to write something to the debug window use the command \texttt{os\_debug}. It can be used like the \texttt{snprintf}()-method you probably know from C.

As you can see the \textit{iSenseDemoApplication} implements several interfaces and consequently, the following methods must be implemented as prescribed by these interfaces:
• stand_by (void), hibernate (void) and wake_up (bool memory_held) inherited from isense::SleepHandler
• button_down( uint8 button ) inherited from isense::ButtonHandler
• confirm (uint8 state , uint8 tries , isense::Time time) inherited from isense::Sender

```cpp
#include <isense/application.h>
#include <isense/os.h>
#include <isense/dispatcher.h>
#include <isense/radio.h>
#include <isense/task.h>
#include <isense/time.h>
#include <isense/button_handler.h>
#include <isense/sleep_handler.h>
#include <isense/modules/pacemate_module/pacemate_module.h>
#include <isense/util.h>

#define MILLISECONDS 1000

using namespace isense;

class iSenseDemoApplication :
    public isense::Application,
    public isense::Receiver,
    public isense::Sender,
    public isense::Task,
    public isense::TimeoutHandler,
    public isense::SleepHandler,
    public ButtonHandler
{
    iSenseDemoApplication(isense::Os& os);
    virtual ~iSenseDemoApplication() ;

    ///From isense::Application
    virtual void boot (void) ;

    ///From isense::SleepHandler
    virtual bool stand_by (void) ; // Memory held

    ///From isense::SleepHandler
    virtual bool hibernate (void) ; // Memory not held

    ///From isense::SleepHandler
    virtual void wake_up (bool memory_held) ;

    ///From isense::ButtonHandler
    virtual void button_down( uint8 button );

    ///From isense::Receiver
    virtual void receive (uint8 len, const uint8 * buf, uint16 src_addr,
```
uint16 dest_addr, uint16 lqi, uint8 seq_no, uint8 interface);

///From isense::Sender
virtual void confirm(uint8 state, uint8 tries,
isense::Time time);

///From isense::Task
virtual void execute(void* userdata);

///From isense::TimeoutHandler
virtual void timeout(void* userdata);

private:
);

ঙździcedesenceDemoApplication::
iSenseDemoApplication(isense::Os& os)
: isense::Application(os)
{
}

ঙździcedesenceDemoApplication::
˜iSenseDemoApplication()
{
}

void
iSenseDemoApplication::
boot(void)
{
    os_.debug("App::boot");
    //os_.allow_sleep(false);
    //os_.add_timeout_in(Time(MILLISECONDS), this, NULL);
}

bool
iSenseDemoApplication::
stand_by (void)
{
    os_.debug("App::sleep");
    return true;
}

bool
iSenseDemoApplication::
hibernate (void)
{
    os_.debug("App::hibernate");
    return false;
}

void
iSenseDemoApplication::
wake_up (bool memory_held)
{
    os_.debug("App::Wakeup");
}
void iSenseDemoApplication::
button_down( uint8 button )
{
}
//-------
void iSenseDemoApplication::
execute( void* userdata )
{
}
//-------
void iSenseDemoApplication::
receive (uint8 len, const uint8 * buf, uint16 src_addr, uint16 dest_addr, 
uint16 lqi, uint8 seq_no, uint8 interface)
{
}
//-------
void iSenseDemoApplication::
confirm (uint8 state, uint8 tries, isense::Time time)
{
}
//-------
void iSenseDemoApplication::
timeout( void* userdata )
{
  os_.add_task( this, NULL);
  os_.add_timeout_in(Time(MILLISECONDS), this, NULL);
}
//-------
/**
 * Source $Source: $ 
 * Version $Revision: 1.24 $ 
 * Date $Date: 2006/10/19 12:37:49 $ 
 * Log$ 
 */

Listing 1: iSenseDemoApplication

4. Extending *iSenseDemoApplication*: TogglingLEDApplication

To implement own applications go to the folder isense_sdk\iApps and create a copy of the *iSenseDemoApplication* - folder. Rename the copy into *TogglingLEDApplication* for example. Now go to your new *TogglingLEDApplication* -folder, open the *.project*-file in an editor and replace the following two lines in there:
(a) replace

\[
\text{\texttt{<name>\textit{iSenseDemoApplication</name>}}}
\]

by

\[
\text{\texttt{<name>\textit{ToggleLEDApp</name>}}}
\]

(b) replace

\[
\text{\texttt{\textbackslash <value>\textit{iSenseDemoApplication</value>}}}
\]

by

\[
\text{\texttt{\textbackslash <value>\textit{ToggleLEDApp</value>}}}
\]

Now open Eclipse. Go to Project Explorer and open the context menu with a right click. Click on Import..., then on Existing Projects into Workspace and afterwards on Next.

Now select isense_sdk\iApps\ToggleLEDApplication as root directory. Check the box ToggleLEDApp and click on Finish.

A folder named ToggleLEDApp appears in the Project Explorer. Go to src and you will find an iSenseDemoApplication.cpp file in there which serves as the starting point for your new application. Now you can extend the iSenseDemoApplication. Listing 2 shows such a possible extension.

There you see two new variables bool led_on and CoreModule* cm_. Using isense::CoreModule you have the opportunity to access functionality of the core module of your iSense node. Note that you need to include

\[
\text{\texttt{<isense/modules/core_module/core_module.h>}}
\]

This requires that your iSense distribution supports the CoreModule functionality (ISENSE_ENABLE_CORE_MODULE is defined in isense_sdk\iSense\ src\isense\config.h. If this is not the case configure a new iSense SDK with Webcompile as it is described above and ensure that you choose the Core Module functionality.

Have a closer look at the execute-method:

\[
\text{cm_.led_on();}
\]

switches the LED of the core module on while

\[
\text{cm_.led_off();}
\]

switches the LED off. The command

\[
\text{os_.add_task_in(Time(1000), this , NULL);}\]
causes the LED to blink (toggle) every second.

```cpp
#ifndef _INC_ISENSE-toggler
#define _INC_ISENSE-toggler

#include <isense/application.h>
#include <isense/os.h>
#include <isense/dispatcher.h>
#include <isense/radio.h>
#include <isense/task.h>
#include <isense/timeout_handler.h>
#include <isense/isense.h>
#include <isense/uart.h>
#include <isense/dispatcher.h>
#include <isense/time.h>
#include <isense/button_handler.h>
#include <isense/sleep_handler.h>
#include <isense/util.h>
#include <isense/modules/core_module/core_module.h>

const uint8 UART_CHANNEL_OUT = 106;

using namespace isense;

class iSenseDemoApplication : 
public isense::Application, 
public isense::Receiver, 
public isense::Sender, 
public isense::Task, 
public isense::TimeoutHandler, 
public isense::SleepHandler, 
public ButtonHandler 
{
public:
 iSenseDemoApplication(isense::Os & os); 
virtual ~iSenseDemoApplication() ;

///From isense::Application 
virtual void boot (void) ;

///From isense::SleepHandler 
virtual bool stand_by (void) ; // Memory held

///From isense::SleepHandler 
virtual bool hibernate (void) ; // Memory not held

///From isense::SleepHandler 
virtual void wake_up (bool memory_held) ;

///From isense::ButtonHandler 
virtual void button_down( uint8 button );

///From isense::Receiver 
virtual void receive (uint8 len, const uint8 * buf, uint16 src_addr, 
uint16 dest_addr, uint16 lqi, uint8 seq_no, uint8 interface) ;

///From isense::Sender 
virtual void confirm (uint8 state, uint8 tries, isense::Time time) ;

///From isense::Task 
virtual void execute( void* userdata ) ;

#endif
```

// From isense::TimeoutHandler
virtual void timeout( void* userdata ) {
    private:
        uint8 y_;
    bool led_on_;  CoreModule* cm_;
};

iSenseDemoApplication::
iSenseDemoApplication(isense::Os& os) : isense::Application(os), y_(24), led_on_(false), cm_(new CoreModule(os)) {
}

iSenseDemoApplication::
~iSenseDemoApplication() {
}

void
iSenseDemoApplication::
boot(void) {
    os_.debug("TogglingLEDApplication");
    os_.add_task_in(Time(1000), this, NULL);
    os_.allow_sleep(false);
}

bool
iSenseDemoApplication::
stand_by (void) {
    TRACE("iSenseDemoApplication: Standby");
    return true;
}

bool
iSenseDemoApplication::
hibernate (void) {
    TRACE("iSenseDemoApplication: Hibernate");
    return false;
}

void
iSenseDemoApplication::
wake_up (bool memory_held) {
    TRACE("iSenseDemoApplication: Wakeup");
}

void
iSenseDemoApplication::
button_down( uint8 button ) {

void
iSenseDemoApplication::
execute(void* userdata)
{
    if(led_on_)
    {
        cm_->led_off();
        led_on_=false;
    }
    else
    {
        cm_->led_on();
        led_on_=true;
    }
    os_.add_task_in(Time(1000), this, NULL);
}

void
iSenseDemoApplication::
receive(uint8 len, const uint8* buf, uint16 src_addr, uint16 dest_addr,
        uint16 lqi, uint8 seq_no, uint8 interface)
{
    os_.debug("receiving");
}

void
iSenseDemoApplication::
confirm(uint8 state, uint8 tries, isense::Time time)
{
    TRACE("iSenseDemoApplication: Confirm");
}

void
iSenseDemoApplication::
timeout(void* userdata)
{
    ...
}

isense::Application* application_factory(isense::Os& os)
{
    return new iSenseDemoApplication(os);
}

Listing 2: TogglingLEDApplication

5. Compiling iSenseDemoApplication

iSense supports multiple platforms. To compile an application for a specific
platform you need a make target for it. You will find an instruction on how to create make targets and binaries in Section 3.3.

**Simulating iSense applications using Shawn** This section describes how to compile iSense-applications for Shawn and run simulations with iSense/Shawn – the internal Shawn-version that is provided by iSense – afterwards. Therefor just follow the steps listed below.

1. Open Eclipse

2. To compile an application for Shawn go to the Make Targets-View within Eclipse and double-click on the appropriate make target (here: shawn).

3. Open the file configuration.jshawn in your application folder for parameterizing and adapting the flow of simulations. The provided iSenseDemoApplication-folder already contains an example-JShawn-configuration-file. Have a look at it at Listing 3.

   At the beginning of the given configuration.jshawn file some simulation parameters are declared. In the part *Execution* you will recognize the already introduced instructions *random_seed*, *prepare_world*, *rect_world* and *simulation*.

   You may modify configuration.jshawn to your needs as you like to, but note that you have to keep hold of the declaration

   ```
   String procs = "isense";
   ```

   Generally modifying configuration.jshawn follows the same rules as already described in the Section 3.2 (JShawn). So, if you need guidance on how to use JShawn, please have a look there.

4. Run simulations by executing the file run_simulation.bat which should also be located in isense_sdk\iApps\iSenseDemoApplication or your own application-folder.

```java
//---Simulation parameters
String em = "simple";
String cm = "disk_graph";
String procs = "isense";

//
int iterations = 6 * 60 * 60;
int width = 200;
int height = 200;
int range = 50;
int nodecount = 50;

// The random seed. Use something greater or equal zero to enable the seed
```
int seed = -1;

//------------------------------------------------
// Execution
//------------------------------------------------
long startTime = System.currentTimeMillis();

//Set the seed
if( seed > 0 )
  shawn.runCommand("random_seed", "action=set seed=" + seed);

//Create a new simulation world and set its parameters
shawn.runCommand("prepare_world","edge_model=" + em +
  " comm_model=" + cm +
  " range=" + range +
  " size_hint=" + range +
  " transm_model=reliable immediate_delivery=true");

shawn.runCommand("rect_world", "width="+width+" 
  height="+height+" 
  count="+nodecount+ 
  " processors=" + procs);

//Run the simulations
shawn.runCommand("simulation", "max_iterations="+iterations);

long endTime = System.currentTimeMillis();
long executionTimeMs = endTime - startTime;
System.err.println("Execution time: " + (executionTimeMs/1000) + " secs");

Listing 3: Provided JShawn-configuration-file for iSense/Shawn

The following Chapter describes how to run an iSense application on real hardware instead of simulating it with Shawn. A detailed approach is explained here.

Transition from simulations to iSense hardware

This Section shows how to install executable iSense applications on the iSense hardware. First the available iSense hardware modules are presented. Subsequently it is illustrated how to generate binaries, download them to the hardware, and monitor and debug the application in the iShell.

iSense hardware iSense is a modular hardware and software platform for wireless networks that is intended for both industry and research applications. The hardware is arranged around a core module that provides computation and communication to each sensor node. It is equipped with an IEEE 802.15.4 [IEEb] compliant radio, a 32-bit RISC controller running at 16MHz, 96kbytes of memory, a highly accurate clock and a switchable power regulator.

It can be combined with a number of sensor modules (including a thermometer, a light sensor, an accelerometer, a passive infrared sensor, a magnetometer for vehicle detection and a camera), different power sources (in-system rechargeable lithium-ion secondary battery, ultra compact 1/2 AA primary battery and traditional AA battery holders), a gateway and I/O module as well as various others. The hardware
is supplemented with a modular operating and networking firmware that is based on object oriented programming. Like this, application specific sensor nodes can be constructed just by plugging together hardware modules containing the required functionality.

Available hardware modules are:

- **Core Module**
  
The iSense core module provides the basis of the iSense modular hardware platform for all kinds of wireless sensor networking applications. Details:
  
  - IEEE 802.15.4 compliant radio, 250 kbit/s, hardware AES Encryption
  - 32 Bit RISC Controller, 16MHz
  - High accuracy (20ppm) real time clock
  - Software controllable voltage regulator
  - Expansion connectors for all kinds of other modules and energy sources
  - SMA connector or integrated ceramic antenna

iSense gives way to high performance and low power sensor networks. Its JN5139 wireless controller provides superior computational capabilities, and offers a large number of peripheral interfaces including I²C, SPI, a 4 channel 11-bit ADC, two 10-bit DACs, two UARTs etc.

In addition a IEEE 802.15.4 compliant, Zigbee-ready radio is included, offering high data rates at ranges of up to 500m while providing hardware AES encryption. The core module also comprises a highly accurate hardware clock that enables precisely timed sleep and wakeup periods while requiring only infrequent resynchronization. The switchable voltage regulator combines high energy efficiency with a wide supply voltage range: it adapts voltages between 1.8 V and 5.5V but can be bypassed for increased efficiency if voltages between 2.2V and 3.6V are available.

- **Gateway Module**
  
The iSense Gateway Module provides connection to other systems such personal computers using USB and RS-232. It enables data exchange as well as serial programming of connected core modules. The USB connector can also be used to power other attached iSense modules, including the Rechargeable Battery Module. In addition, the Gateway Module provides 3 LEDs, 2 buttons and a potentiometer.

- **Measurement Module**
  
The iSense Measurement Module provides convenient access to all pins of the 34 pin expansion connector. It is intended for signal measurements, for debugging and rapid sensor board development. On the one hand it can be used to track hardware behavior as well as software functionality by measuring signals with
an oscilloscope. On the other hand, new sensors can be quickly attached to the iSense platform by just plugging them to the measurement board.

It provides access to both UARTs, the DACs and the ADC, the SPI and I²C bus, supply voltage and ground, the reset signal, as well as to 9 general purpose I/O pins.

- **Battery Modules**

Three different battery modules are available for the iSsense hardware platform.

- The iSense AA Battery Module accommodates two standard AA batteries for high capacity and low cost sensor network applications.
- The iSense 1/2AA Battery Module comprises a 1/2AA battery holder as well a digital battery monitor. It gives rise to ultra compact setups that allow precise battery state monitor, providing accurate information on expended and remaining energy.
- The iSense Rechargeable Battery Module combines a digital battery monitor, a charge controller and a high capacity lithium-ion rechargeable battery. This module enables in-system charging just by connecting the system to a wall mount adapter, significantly easing the handling of battery powered sensor systems. The battery monitor gives provides precise information on the energy currently stored in the battery, accumulating both charging and discharging cycles.

- **Vehicle Detection Module**

The iSense Vehicle Detection Module is based on an anisotropic magneto - resistive (AMR) sensor that is combined with two cascaded amplifier stages and additional control and compensation circuits. In combination with the analogue-digital converter of the iSense Core module it can be used to detect large metal objects such as cars moving by.

- **Security Sensor Modules**

The iSense Security Sensor Module series features a passive infrared (PIR) sensor and/or a 3-axis accelerometer.

The PIR Sensor can be used to detect moving objects that feature a temperature different from the environment (such as humans) in distances of up to 10 meters. The sensor offers a wide range of 110° for comprehensive monitoring.

The 3-axis accelerometer can be configured to cover accelerations of ± 2g or ± 6g. In addition to delivering acceleration values via the I²C digital interface, it can generate interrupts on movement, direction change or free fall.

In addition, a camera module that can take color pictures with a mega pixel resolution can be attached. The images are preprocessed, so they can be scaled down to lower resolutions and compressed according to the JPEG standard.
Camera Module Available variants are equipped with

- PIR sensor,
- Accelerometer or
- both sensors.

- Environmental Sensor Module

The iSense Environmental Sensor Module combines a thermometer and a light sensor for environmental monitoring. Both sensors are accessed via the I²C serial interface.

The thermometer provides a configurable interrupt threshold value as well as a hysteresis value. Like this it can wake up the device if given temperatures are exceeded.

The light sensor provides two light values, one delivered by a sensor sensitive to all kinds of light, and another by an infrared light sensor. Their difference yields the luminance considering human visible light only.

**Device Programming**  The easiest way to program nodes with an application is to use a Gateway Module. First plug together the Core Module and the Gateway Module. To connect the sensor node and the PC use the USB connector of the Gateway Module. You can also use Battery Modules instead of the Gateway Module, but note that you have to check the battery status to verify that the hardware is functioning correctly.

**Generating executable files for iSense nodes**  Before you can start programming your sensor nodes you have to produce executables of your application. Therefore create make targets in Eclipse. Start Eclipse and go to Window -> Show View -> Make Targets. Have a look at Figure 75 to see how the settings for a Jennic JN5139R device would be for example. For a more detailed instruction look at Section 3.1.

Now you are able to compile your application and produce the needed binaries by clicking on the according make target for your hardware platform.

To get a connection between your PC and a Gateway Module you have to install the iSense Gateway Module USB Driver on your system:

**Installing the iSense Gateway Module USB Driver for Windows**

1. Go to http://www.ftdichip.com/Drivers/VCP.htm
2. Click on setup executable (in Comments column at right)
3. Install the driver

Finally install and start the iShell to program your nodes.
Using the iShell  First install iShell. Thereto download iShell from


Execute ishell.exe.

If iShell is running first click on the iShell preferences button to edit the preferences. Choose the COM-port your sensor node is connected to an click "Ok" in the iShell preferences menu (Figure 76) afterwards.

Then a message New connection to Jennic device at COM4 should appear. Now activate the tab Flash Loader (Figure 77) and ensure that the checkbox Trigger programming mode is checked. Search and select the binary of your application that must have been compiled for the hardware platform you are using (e.g. \isense_sdk\iApps\iSenseDemoApplication\bin\JN5139R\iSenseDemoApp.bin) after clicking on the open folder-button you can see on the right of the Flash Loader
menu. Now click Start to program your sensor node.

![iShell Flash Loader](image)

**Figure 77: iShell Flash Loader**

**Monitoring of iSense nodes** If the programming process finished successfully go to tab **Serial Monitor** (Figure 78). There you see a monitor window where you can follow the application’s behavior with the help of debug messages. Furthermore there are three buttons. With the first button which looks like a brush you can clear the monitor window. With the second button you are able to pause or continue the application and with the third button reboot the sensor node.

You have the opportunity to activate other plugins. Therefor go on tab **Plugins** (Figure 79) and activate the according checkbox of the desired plugin. The chosen plugin tab will appear promptly.

**System and Hardware Verification** This Section describes how to ensure that possibly occuring bugs will not be a consequence of not working hardware or tools. That way you save hours of searching for bugs.

- **PC Tools Verification:**
  
  Ensure that the USB driver is installed on your system as well as the iShell. Also verify that the the checkbox **Trigger programming mode** in the **Flash Loader**-tab is checked.
**Figure 78:** iShell Serial Monitor

**Figure 79:** iShell Plugins
Hardware Verification:
Verify that the on/off switch of the sensor node is switched to ”on”. It is ”on” if the switch is adjusted to the opposite side of the antenna. If you are using batteries check if they are fully charged. Verify that the voltage of the batteries does not fall during electrical loading. Ensure that Gateway Module and PC are accurately connected.

Troubleshooting This Section describes possible error messages and solutions of the responsible problems.

- `java.io.IOException: Bad file descriptor in nativeavailable`
  Solution: ensure that sensor node and PC are connected well.

- `Supplied device is null. Using Null device Error on new Jennic device connection :gnu.io.NoSuchPortException`
  Solution: ensure that sensor node and PC are connected well. Also check if you have chosen the right COM-port in the `iShell Preferences menu`.

- After starting to program a sensor node following error message occurs: `Timeout while waiting for data(...) Still waiting for a connection`
  Solution: ensure that you have checked the box `Triggering programming mode`. Afterwards retry to program the sensor node.

- After starting to program your device the following error message occurs: `Exception while waiting for connection Still waiting for a connection`
  Solution: check if you have chosen the right COM-port in the `iShell Preferences menu`.

- Other possible solutions for problems that disrupt your programming process immediately after starting it:
  
  - If you are using a battery module check if the batteries you are using are fully charged and do not de-energize while using resistances.
  
  - Check that you have compiled your application for the right device and check that you have chosen exactly the right compiled version of your application for the device you are using as source path in the `Flash Loader` menu.

- Flashing the nodes sometimes fails somewhere in the middle. It works fine on the second try.

  Solution: Probably the program that is already running on the node crashed, and hence the watchdog is not set back periodically. Hence the device is reset by
the watchdog after some time, then hangs up again, and so on. When you start programming at a certain point in time, the watchdog may reset the device while programming. To prevent that, reset the device before starting the programming.

- The command `make` was not found

  Solution: Go to www.cygwin.com. Download and run the `Setup.exe`. In the section `Devel`, search for `make`. If the entry of `make` is set to `skip`, change it to `install` by clicking on `skip` until a version number is visible.

**Running tests on the FRONTS testbed**

This Section gives an outlook on the FRONTS testbed that will be built up in Lübeck in the next months. It will be remotely accessible for all project partners. Opportunities for programming and monitoring the FRONTS testbed’s sensor nodes are described below.

Programming and monitoring of iSense sensor nodes is feasible by using the tool `iShell` that is described in Section 3.3. To be able to use remote functionalities with the iShell the actual version of the tool will be upgraded soon.

Thus by upgrading the iShell instead of developing a new tool from scratch users that got used to program and monitor their sensor nodes with the actual version of the iShell will be able to easily habituate to the new iShell and its new remote function. Nevertheless it will be possible to use the new iShell remotely as well as locally as before.

Another main advancement will be the opportunity of monitoring more than one sensor node in only one iShell.

The FRONTS sensor node testbed will consist of Core Modules, Gateway Modules with USB connectors, Security Sensor Modules with accelerometer where some of these modules will hold extra PIR sensors, Battery Modules with lithium-ion rechargeable batteries, power supply packs and antennas with SMA port.

The technical committee will decide about the final configuration of the testbed’s topology.

At this point of time the sensor node’s arrangement that is presented in Figure 80 shall act as suggestion for a possible topology. The image shows the Institute of Telematics of the University of Lübeck. The red dots stand for the sensor nodes placed in the institute’s offices. The blue markings highlight the availability of electricity.
Figure 80: Suggestion for a possible testbed’s sensor node topology.
4 Coding Style

This section contains the general coding guidelines for Shawn. The following lines illustrate the basic idea of our coding style by a simple example class, but the next sections also describe our style in more detail.

```cpp
class ClassNameMixedUpperLower
{
public:
    virtual ~ClassNameMixedUpperLower();
    // always a virtual destructor, some compilers dont work
    // otherwise
    int methods_are_stl_like( void ) throw();
    // access to stored stuff is provided via two methods, const
    // and writable.
    // latter is indicated by _w suffix
    // get_ is omitted (so usual pair is set_value(value),
    // value(void)
    const SomeThing& some_thing( void ) const throw();
    SomeThing& some_thing_w( void ) throw();
    void set_some_thing( SomeThing& ) throw();
private:
    int member_variables_have_an_underscore_;
};
```

4.1 General Style

This section describes the Look & Feel of our source code. It contains the intending, the placement of brackets, when and where to set single spaces, and the maximal line width.

Indenting

Indenting can be done either by spaces or by tabs, but consistently per file. So do not mix spaces and tabs in an existing file. Instead, use what the predecessor has used. If you use spaces, indent with exactly three spaces.

Brackets

Brackets are consistently written in separate lines, and thus never written behind a statement. This is done for conditional expressions, loops, classes, and namespaces.

```cpp
if ( a == b )
{
    // do something
}
else
{
    // do something else
}

for ( int i = 0; i < 5; i++ )
{
    if ( foo == i )
    // do something
}
```
When writing one-liners like
\[
\text{for ( int } i = 0; i < 5; i++ )
\]
\[ j++; \]
brackets are not mandatory (albeit they can be used), but if there are nested one-liners, one should use brackets for a better readability. So better write
\[
\text{if ( a == b )}
\]
\[
\{ 
\text{for ( int } i = 0; i < 5; i++ )
\}
\]
\[ j++; \]
}\]
instead of leaving the brackets out.

Spacing

To enhance readability, spaces are used to separate declarations (after a comma), mathematical expressions, and before or after parenthesis.

For example, a function header looks like this:
\[
\text{void foo( int a, int b );}
\]
\[
\text{void bar( void );}
\]

An example for a conditional expression may be:
\[
\text{if ( a == b )}
\]

And a mathematical expression should be written as follows:
\[
c = d + 4;
\]

Wrapping

Line wrapping is done after at most 80 characters to enhance readability and portability. Note that there are also users who work in text-based environments.

If a mathematical expression must be wrapped, split it regarding to the context of the expression:
\[
\text{if ( ( a == b ) }
\]
\[
\&\& ( c > d || e < f
\]
\[
\| d > e )
\]
\[
\&\& ( g != h )
\]
Separators

Separators are used to enhance readability. They can be used either inside of methods to group correlated statements, or outside of methods to distinguish one method against the other.

**Inside Methods**  Group correlated statements together, and separate these groups by newlines. This enhances readability because it enables reading block by block in logical sections. Moreover, the general view is improved by doing so. Have a look at the following examples.

```cpp
const SimpleAppMessage* msg = dynamic_cast<const SimpleAppMessage*>(mh.get());
if( msg != NULL )
{
    neighbors_.insert( &msg->source() );
    if( owner() != msg->source() )
    {
        INFO( logger(), "Rcvd message from" << msg->source().label() << ":" );
    }
    return true;
}
return Processor::process_message( mh );
```

```cpp
TransmissionModel::reset();
while( !aired_messages_.empty() )
{
    MessageInfo* mi = aired_messages_.front();
    aired_messages_.pop();
    delete mi;
}
```

**Outside Methods**  To separate one method implementation against the other, there is a delimiter between any two methods as shown in the following source.

```cpp
 [...] // --------------------------------------------------
 void MyClass::
    my_function( void )
    throw()
{
    [...] // --------------------------------------------------
 void MyClass::
    my_other_function( void )
    const throw()
{
    [...] // --------------------------------------------------
 [...]```
4.2 Naming

This section contains rules for naming identifiers like variables and methods, and filenames. There are rules for both selecting names and spelling names.

Selection of Names

When you think about names for identifiers, choose meaningful ones. This enhances readability, and suppresses the need for additional comments.

Hence, do not write the following:

```
int t; // time of last reception
```

Instead, write it as follows:

```
int time_of_last_reception;
```

Then, use only comprehensible abbreviations for names. It is valuable to use abbreviations like `idx` for index, or `cnt` for count. But do not abbreviate a sentence like `time of last reception` to `tolr`.

In general, http://www.objectmentor.com/resources/articles/naming.htm is a mentionable article that describes naming in detail and is worth reading.

Spelling of Names

Methods and Parameters Methods and parameters are completely written in lower case characters ('a'..'z', '0'..'9', '_'). They begin with a letter ('a'..'z'). Multiple words are separated with underscores. Examples are:

```
void set_node_label( int );
void do_something( void );
int time_of_last_reception;
int node_cnt;
```

Class Names Classes are written in mixed upper and lower case characters ('A'..'Z', 'a'..'z', '0'..'9'). They begin with an upper case letter ('A'..'Z'), followed by lower case ones. Multiple words are separated by a new upper case character. If a word is a known abbreviation (like 'MAC Layer', for example), only the first character is written as an upper case one. Examples are:

```
class MacLayer;
class SimulationController;
class HelloWorldProcessor;
```

Prefixes and Suffixes

Getters and Setters If class members can be accessed (per set or get) via methods, these methods follow a certain naming rule. All setters have the prefix `set_*:`
void set_some_thing( SomeThing& obj ) throw();

Getters do not have any prefix. To distinguish writable and constant getters, writable ones get the suffix *_w:

const SomeThing& some_thing( void ) const throw();
SomeThing& some_thing_w( void ) throw();

Private Member  Private members of classes always end with an underscore:

int private_member_of_a_class_

As a side effect, variable names in setters and constructors can easily be chosen.

void set_value( int value )
{
    value_ = value;
}

Files and Folders

Sourcefiles are named after the particular class (e.g., the files simulation_controller.* contain the class SimulationController). Like naming methods and parameters, file-names are completely written in lower case characters ('a'..'z', '0'..'9', '_'). They begin with a letter ('a'..'z'). Multiple words are separated with underscores.

File extensions are *.h for headers, and *.cpp for source files.

When creating directories, there are the same rules as for files. Use only lower case characters or digits, separate by using the underscore, and do not use spaces.

4.3 Programming Behavior

Using Macros

The usage of #define should be avoided, because there are enough alternatives available in C++ (const (const int foo = 0), inline functions, templates), and macros also do not obey scopes or type rules.

So do not use macros unless it is absolutely inevitable. Especially, avoid using macros in header files. If at all, use them in source files where they are only applied strictly local.

Using the Range of the Language

Make yourself familiar with the programming language C++, and use available constructs instead of reinventing the wheel. Especially, make use of the STL, their containers and iterators. Use streams and strings. If something is worth made generic, use templates.
4.4 OOP

Destructor

Always write virtual destructors:

```cpp
virtual ~Class();
```

Const Member Functions

If a method of a class does not modify the state of an object, declare it as const. As const objects are often used in Shawn, this increases usability for developers.

Exception Specification

Each method of a class must specify the type of potentially thrown exceptions. If no exceptions are thrown, use the empty specification. Examples are:

```cpp
const ProcessorKeeper& processor_keeper( void ) const throw();
ProcessorKeeper& processor_keeper_w( void ) throw();
virtual void run( SimulationController& ) throw( std::runtime_error ) = 0;
```

4.5 Comments

Language

All comments and all documentation are written in English, because Shawn is used in an international environment.

Header Files

Header files are completely documented in doxygen-style to be able to automatically generate documentation in HTML, pdf, or the like. See http://www.stack.nl/~dimitri/doxygen/manual.html, for example, for details. Thereby the class itself, each method, and each parameter must be described.

The description of the class starts with a brief description of what the class is supposed to do. This is an one-liner that is used together with the doxygen keyword `\brief`. Then, a detailed description follows. It contains the general behavior of the class, the context in which it is used, potential usage hints (e.g., with examples), and so on.

Next, methods must be described. If suitable, they are grouped with the aid of the doxygen keywords `@name`, `@{`, and `@}`. Then, for each function the purpose and usage are documented. Optionally, the description can be started with an one-liner and the keyword `\brief`. At last, all parameters (if there are any) and the return value (if not void) are documented.
Then, the class members are documented, generally with a short one-liner that describes the purpose of the value and in which context it is used.

However, here is a skeleton of a general header file:

```cpp
/** \brief Short Description (one line) 
 * Detailed description (multiple lines). */
class DoxygenExample {
    public:
        ///@name Constructor/Destructor 
        ///@{
        /** constructor description, optionally with \brief 
         * */
        DoxygenExample();
        /** destructor description, optionally with \brief 
         * */
        virtual ~DoxygenExample();
        ///@}
        ///@name Getters and Setters 
        ///@{
        /** function description, optionally with \brief 
         * \param value brief description */
        void setValue(int value);
        /** function description, optionally with \brief 
         * \return brief description */
        int value(void);
        ///@}
    private:
        /// purpose of value
        int value_;   
};
```

**In-Source Documentation**

In general, give rough descriptions of what you are doing and why. Do not comment obvious lines like

```cpp
a += 1; // add one to a
```

Then, use // for documentation, also for multi-line comments (note that this rule holds only for in-source documentation):

```cpp
// first line of comment ...
// ... followed by the second line.
```
5 Configuration Management

A primary objective of FRONTS is to establish a repository of algorithms and experiments: a central place for storing the software implementations and the corresponding documentation for executing experiments. Each algorithm implemented will be stored in the repository as a separate software component.

As the project will evolve in time, we expect the number of implemented algorithms to grow. Therefore, it is essential to provide a set of tools that will allow us to manage the software developed, organize the documents and keep track of the development effort. For this reason we decided to follow the software engineering approach of Software Configuration Management (SCM) [BSH80], that is, the discipline of identifying the configuration of a system at discrete points in time for purposes of systematically controlling changes to this configuration and maintaining the integrity and traceability of this configuration throughout the system life cycle. SCM ensures that this evolution is efficient and controlled, so that the individual components fit together to form a coherent whole [Whi91].

In simple terms, the goals of Software Configuration Management are to (1) identify change, (2) manage that change, (3) ensure that the change is being properly implemented, (4) report the change to others who may have an interest, and (5) record the change for historical reference. In terms of FRONTS we will apply SCM for technical and administrative direction and surveillance over the life cycle of software components (or configuration items) to:

- Identify and document the functional characteristics of software components.
- Control changes to software components and their related documentation.
- Record and report information needed to manage components effectively, including the status of proposed changes and implementation status of approved changes.
- Make sure every defect has traceability back to the source.
- Facilitate team interactions related to the process.

There exist many software tools for tracking changes and configuration control, both commercial and open source. The majority focus on particular aspects of SCM like version control, status accounting, project management or issue tracking. Some popular tools that support SCM are Subversion (SVN) [Col00], a version control system for recording changes, Bugzilla [ZZba], a issue tracking system designed to help you manage software development and Buildfactory [ZZbb], a project lifecycle management tool. There also exist integrated environments that offer a collection of tools for applying SCM to software projects. These systems are also known as integrated collaboration tools. Most notable examples of such systems are G-Forge [Zgf], the system used by SourceForge.org, Microsoft’s Team Foundation Server [Zmi], IBM’s...
Rational ClearCase [Zib] and Borland’s StarTeam [Zbo]. The benefit of using a collection of standalone tools is that each tool is sophisticated and highly specialized. On the other hand, the integrated collaboration systems may include individual tools that are less sophisticated but all tools are already integrated and no further effort is required by the system administrator. For an almost complete list of available software tools and integrated solutions for supporting SCM, see [Zcm].

In FRONTNS we decided to use TRAC [Ztr], an integrated system for supporting SCM. A key criteria for our selection was that TRAC is an open source system and does not require the provision of expensive licenses. It is very stable and has an active community that continuously improves its performance and extends its capabilities. Furthermore, it uses a minimalistic approach that simplifies the overall procedure and reduces the overhead of applying complex management techniques. The most central features are an enhanced wiki, a basic issue tracking system and a bug-tracking system over a web-based user interface. TRAC is reported to have more than 450 installations worldwide. Among the users of TRAC is NASA’s Jet Propulsion Laboratory, which reports that it uses this tool to manage various deep space and near space projects. Another project that uses TRAC is WebKit, the open source browser rendering engine that powers Apple’s Safari browser (and also Adobe’s AIR among other implementations).

5.1 Status Accounting

On any team project, a certain degree of confusion is inevitable. The goal is to minimize this confusion so that more work can get done. A basic procedure of SCM in order to coordinate software development and minimize confusion is status accounting. That is recording and reporting the status of components and change requests, and gathering vital statistics about components. TRAC provides the ability to evaluate the progress of the development of each software components and report the current status.

In FRONTNS we consider each algorithm as a separate software component. We are therefore interested to evaluate at regular time intervals the progress of each individual component (algorithm) separately. When a new algorithm is about to be added to the repository, the developer must contact the administrator of TRAC in order to introduce the new component in the integrated SCM environment. Each algorithm must have a name that does not contain any white space and non-latin characters. After introducing the component, its name cannot be changed.

Based on the Technical Annex, there are three different ways to contribute to the project’s repository:

1. Each group works on their own specific software/hardware
2. They work on the common simulation toolkit
3. They work on the common experimental platform.

Based on this differentiation and given the development cycle encouraged by Shawn (see Fig. 2, pp. 10), we consider the following versions for each component:

- Version A – Implementation of component in specific software (e.g. TOSSIM).
- Version B – Centralized implementation of component in Shawn.
- Version C – Simplified distributed implementation of component in Shawn.
- Version D – Fully distributed implementation of component in Shawn.

TRAC allows to associate the changes made to each component and corresponding files to a specific version. It also provides a wiki system for creating pages for documenting the progress of a component. These pages can be associated with a particular file and a specific version.

We consider a milestone for the repository when the implementation of an individual component is complete. Given the above versioning, the implementation is complete when all five versions are finalized. Thus, for each component introduced in TRAC the administrator will associate one milestone that signifies the completion of the implementation. TRAC offers the possibility to view in a single page the status of the milestones of all the components.

This is a simple approach that provides a very basic level for status accounting. This selection allows each developer to decide individually how much details to provide for the progress of her components. Although uniformity will be difficult to reach (above the basic status accounting) it guarantees that SCM will not become a bureaucratic roadblock that impedes the work. While SCM is required based on the Technical Annex of the project, it is crucial to truly apply it in FRONTS since we strongly believe that it will assist researchers in controlling and tracking their work, while ensuring that nothing is lost or destroyed.

5.2 Version Control

Version control server software (also known as Revision Control) provides a convenient way to maintain current and historical versions of files such as source code, web pages, and documentation. Changes to these files are usually identified by incrementing an associated number or letter code, termed the “revision number”, “revision level”, or simply “revision” and associated historically with the person making the change. A simple form of revision control, for example, has the initial issue of a drawing assigned the revision number “1”. When the first change is made, the revision number is incremented to “2” and so on.
In FRONTS we decided to use Subversion (SVN) [Col00], a version control system initiated in 2000 by CollabNet Inc. Subversion is fully compatible with TRAC. Subversion is well-known in the open source community and is used on many open source projects such as: Apache Software Foundation, KDE, GNOME, Free Pascal, FreeBSD, GCC, Python, Django, Ruby, Mono, and previously also Samba (which now uses git). SourceForge.net and Tigris.org also provide Subversion hosting for their open source projects. Google Code and BountySource systems use it exclusively. Subversion is released under the Apache License, making it free software.

SVN provides a log of all changes made on files and offers the ability to easily revert to any version. While backups provide something like this, their granularity is usually relatively low—at most once a day, and often less. With version control, it is easy to decide how much change is worth committing a new version (usually, one work session correlates to one new version), and it is always possible to come back and revert to a specific version in one simple command. Moreover SVN is a very easy way to centralize information.

The Subversion filesystem can be described as a three dimensional filesystem (see Fig. 81). Since most representations of a directory tree (e.g., tree view) are two dimensional, the added dimension is that of revisions. Each revision in a Subversion filesystem has its own root, which is used to access contents at that revision. Files are stored as links to the most recent change; thus a Subversion repository is quite compact. The storage space used is proportional to the number of changes made, not to the number of revisions.

Each Software Component (i.e., algorithm under development) will be stored under a different directory. The name of the directory is the same of the component net (see previous section). It must be unique for each algorithm, it should not include white space characters or any non-latin characters.

The internal structure of the directory of each component must contain the following items:

- doc – a folder where the technical documents of the components will be placed.
- scripts – a folder for storing scripts for executing the component either as a standalone piece of software or through Shawn.
- build.xml – an ANT configuration file [ZZa] for compiling the component and executing simple test runs.
- src – a folder containing all the source files.

Figure 81: The Subversion filesystem
**src/A** – a folder containing all the source files of the implementation of the component in specific software (e.g. TOSSIM).

**src/B** – a folder containing all the source files of the implementation of the component as a centralized component in Shawn.

**src/C** – a folder containing all the source files of the implementation of the component as a distributed component in Shawn.

**src/D** – a folder containing all the source files of the implementation of the component as a fully distributed component in Shawn.

**src/E** – a folder containing all the source files of the implementation of the component compatible with iSense.

The Subversion filesystem uses transactions to keep changes atomic. A transaction is begun from a specified revision of the filesystem, not necessarily the latest. The transaction has its own root, on which changes are made. It is then either committed and becomes the latest revision, or is aborted. The transaction is actually a long-lived filesystem object; a client does not need to commit or abort a transaction itself, rather it can also begin a transaction, exit, and then can re-open the transaction and continue using it. Multiple clients can access the same transaction and work together on an atomic change.

Subversion uses the interfile branching model to handle branches and tags (see Fig. 82). A new branch or tag is created with the `svn copy` command, which should be used in place of the native operating system mechanism. Subversion does not create an entire new file in the repository with its copy. Instead, the old and new files are linked together internally and the history is preserved for both. The copied files take up only a little extra room in the repository because Subversion saves only the differences from the original files.

When the implementation of a particular component (or a particular version) is complete, the code must be tagged under the top-level `tags` folder using the name of the component and the version, e.g., for component “ALGORITHM”, version A, the tag should be “ALGORITHM-A”. This ensures that the final versions of the implementations are clearly marked and available at a central directory.

All the files in each branch maintain the history of the file up to the point of the copy, plus any changes made since. Changes can be merged back into the trunk or between branches. To Subversion, the only difference between tags and branches is that changes should not be checked into the tagged versions. Due to the differencing algorithm, creating a tag or a branch takes very little additional space in the repository.
5.3 Issue Tracking system

An issue tracking system is a software tool that is designed to help programmers keep track of reported issues related to the software in their work. They are used to manage and maintain lists of issues, as needed by a development team. An issue is a file contained within the issue tracking system which contains information about support interventions made by the researchers regarding an incident that is preventing them from using a particular component of the repository.

Each issue has a unique reference number which is used to allow the developers to quickly locate and evaluate the status. It is very common that changes made to the software code of a component to be related to a specific bug or issue. TRAC allows to associate the changes made in Subversion with the specific ID of an issue. Then, when the programmer browses the log messages of Subversion, TRAC automatically creates a hyperlink out of each issue ID in the log message which fires up the browser to the issue mentioned.

In FRONTS each issue (or ticket) contains the following information attributes:

- Reporter - The author of the issue.
- Component - The individual component (algorithm) this issue concerns.
- Type - The nature of the issue. The following issue types are supported:
  1. Task – for issues related to implementation of new features.
  2. Enhancement – for issues related to extensions to existing functionality or to improvements to functionalities implemented in a non-efficient way.
  3. Defect – for issues related to bugs and functionality that does not perform as expected.
  4. Test – for issues related to testing particular functionality of a component.
  5. Simulation – for issues related to conducting simulations on the central simulation testbed.
  6. Experiment – for issues related to conducting experiments on the central experimental testbed.

Figure 82: Visualization of a Subversion project
• Version - Version of the component that this issue pertains to.
• Keywords - Keywords that an issue is marked with. Useful for searching and report generation.
• Milestone - The milestone associated to the individual component this issue concerns.
• Assigned to/Owner - Principal person responsible for handling the issue.
• CC - A comma-separated list of other people or E-Mail addresses to notify. Note that this does not imply responsibility or any other policy.
• Resolution - Reason for why an issue was closed, e.g., when the incident reported is fixed or the enhancement requested is implemented, an issue is said to be closed. The following resolutions are supported:

1. Completed – the issue is resolved completely.
2. Invalid – the issue cannot be resolved (e.g., the bug indicated does not exist).
3. Wont implement – the developer of the component will not implement the issue (e.g., the enhancement proposed is not within the scope of the component).
4. Duplicate – the issue already exists in the database with a different ID.

• Status - The current status of the issue. One of new, assigned, closed, reopened.
• Summary - A brief description summarizing the problem or issue.
• Description - The body of the ticket. A good description should be specific, descriptive and to the point.

TRAC supports the concept of the life cycle for an issue which is tracked through status assigned to the issue. Once a ticket has been entered into TRAC, its attributes can change at any time by annotating it. This means changes and comments to the issue are logged as a part of the ticket itself. TRAC allows developers to move the issue to another status, or delete the issue. An important feature is being able to use wiki formatting in issues descriptions and comments. It also allows links to refer to other issues, changesets or files to make a ticket more specific and easier to understand. When viewing an issue, the history of changes will appear below the main ticket area. The life cycle of an issue and the information about the state transitions are depicted in Fig. 83.

5.4 Integration of Components

A main goal for setting up a central experimental repository is to provide the ability to compare a newly implemented algorithm with those that already exist in the repository. This is an invaluable tool for algorithm designers as they can easily identify the advantages and disadvantages of their approach. Since each algorithm is considered
to be a different software component, to achieve such a goal it is imperative that components must be integrated. Therefore it is essential to provide a common integration procedure.

Continuous Integration [DMG07] is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly. The automated, continuous build increases the productivity.

Continuous Integration emerged in the Extreme Programming (XP) community, and is now considered one very promising tool for effective software engineering. It has many advantages compared to the classical integration approach (once, when implementation is complete):

- When unit tests fail, or a bug is discovered, developers might revert the codebase back to a bug-free state, without wasting time debugging.
- Integration problems are detected and fixed continuously - no last minute problems before release dates.
- Early warning of broken/incompatible code.

Figure 83: The TRAC issue tracking workflow
• Early warning of conflicting changes.
• Immediate unit testing of all changes.
• Constant availability of a “current” build for testing, demo, or release purposes.
• The immediate impact of checking in incomplete or broken code acts as an incentive to developers to learn to work more incrementally with shorter feedback cycles.

Notable examples of continuous integration software include: Continuum [Apa], an enterprise-ready continuous integration server by Apache, Bamboo [ZZaoo], a continuous integration server from Atlassian Software Systems (also makers of JIRA, Confluence and Crowd) and CruiseControl [ZZcet], a free, open source framework for a continuous build process.

In FRONTS we decided to use Hudson [ZZh], an extensible continuous integration engine. Hudson provides an easy-to-use so-called continuous integration system, making it easier for developers to integrate changes to the project, and making it easier for users to obtain a fresh build. The main criteria for selecting Hudson over other continuous integration systems are: (i) it is free, open source software, (ii) it integrates with TRAC and (iii) it can execute Apache Ant based projects, as well as arbitrary shell scripts and Windows batch commands. Hudson is one of the few systems that allows the integration of components developed in Linux-based systems and Windows. This is of particular importance to FRONTS repository since each group may work on their own specific software/hardware (that are not compatible with a single OS). In addition, given that Shawn supports both OS families, we do not wish to force groups to use a particular OS just for integrating components in the repository.

Given a new component is introduced in TRAC, as soon as the developer provides the build.xml file (the ANT configuration file [ZZa] for compiling the component and executing simple test runs) a new job must be created in Hudson. The job name will be set to match the name of the component and associated with the component in TRAC. For uniformity purposes, the build.xml file must support the following commands:

build-A – a command for compiling the component implemented in specific software/hardware.

test-A – a collection of test runs that validate the success of the compilation. Such test runs may include execution of the algorithm in simple topologies and small problem instances.

build-B – a command for compile the implementation of the centralized component in Shawn.
test-B – a collection of test runs for Shawn that aim to validate the success of the compilation using simple topologies and small problem instances. Also execution of valgrind [ZZv] to ensure that such leaks do not occur in our software (see next section).

build-C – a command for compile the implementation of the component as a distributed component in Shawn.

test-C – a collection of test runs for Shawn that aim to validate the success of the compilation using simple topologies and small problem instances. Also execution of valgrind [ZZv] to ensure that such leaks do not occur in our software (see next section).

build-D – a command for compile the implementation of the component as a fully distributed component in Shawn.

test-D – a collection of test runs for Shawn that aim to validate the success of the compilation using simple topologies and small problem instances. Also execution of valgrind [ZZv] to ensure that such leaks do not occur in our software (see next section).

build-E – a command for compile the implementation of the component compatible with iSense.

test-F – a collection of test runs for the iSense hardware. Also execution of valgrind [ZZv] to ensure that such leaks do not occur in our software (see next section).

clean – a command that removes temporary files and compiled objects.

doc – a command that produces documentation for doxygen-based source files (see Chapter 4).

In addition to the ANT build file, the developer may include addition scripts for the execution of the algorithm under specific topologies and for particular problem instances. These scripts must be positioned within the scripts folder (see Version Control). The developer may hook these scripts with Hudson so that they are submitted on demand for execution on the central experiments testbed.

The execution of builds can be started by various means, including scheduling via a cron-like mechanism, building when other builds have completed, and by requesting a specific build URL. Hudsons allows to monitor executions of externally-run jobs, such as cron jobs and procmail jobs, even those that are run on a remote machine. When the jobs are completed, detailed reports on the outcome of the execution of each script is reported on the system web-page and the programmer is notified via e-mail. Hudson keeps those outputs and makes it easy to notice when something is wrong.
6 Software Evaluation

Software development needs a good definition of quality assurance which is especially important for large software projects and long-running embedded application as in the case of wireless sensor networks. Since human beings are prone to produce errors, especially when developing software, we present a number of quality factors to assess the quality of the project’s results. By achieving each of the presented quality factors without exception, we are able to deliver functional, correctly running, and easy to use software.

6.1 Correctness

The Correctness factor evaluates the extent to which the software adheres to its specifications and standards. However, it is hard to evaluate semantic correctness in a general manner. In terms of software development in FRONTS, we therefore define two basic requirements ensuring the correctness of our product to a certain extent:

- All produced source code must compile without any warnings, and
- the source code must not have any memory leaks.

The first requirement, successful compilation without any errors and warnings, uses the compiler for ensuring a minimal amount of correctness. This covers not only syntactical correctness which would generally result in a compilation error, but also a limited amount of semantic correctness which would generally result in compiler warnings as far as the compiler is able to recognize such faults. A reasonable choice is to enable warnings about questionable constructions that are easy to avoid. For instance, when using the well-known Gnu Compiler Collection (GCC, [Fre84]), this implies the usage of the compiler switches –Wall and –Werror.

Memory leaks occur whenever a program requests memory which it does not release when it is no longer needed. Hence, over time, the program consumes more and more memory, and the system eventually runs out of memory. Such leaks are hard to find and can slow down the program’s execution or even lead to its abortion, especially when running for a longer time or on large data. We use valgrind [ZZv] to ensure that such leaks do not occur in our software.

6.2 Efficiency

The Efficiency factor evaluates the resource consumption of a software component. Since different kinds of applications have different characteristics, a general metric is required that can be applied to each of our software components. Such an evaluation measures
- Code size,
- Processing time,
- Memory usage, and
- Number of exchanged messages.

The code size of a software component can be easily obtained using the executables size or by using gcc’s `size` command. The code size must be roughly the same or smaller than similar components. The remaining three factors can be easily collected using Shawn. We measure a software component’s efficiency by running simulations in Shawn for 10, 100 and 1000 nodes and rate its relative scalability.

Processing time and memory usage are taken with the aid of the used operation system. For example, the Unix `time` command can be used to obtain the processing time of a simulation and the Unix `ps` command can be used to query the amount of memory required by a process. The number of exchanged messages can be automatically counted by Shawn.

The obtained number can then be used to evaluate the efficiency of the implementation by comparing them with the expected values of the algorithm’s complexity. For instance, if the run-time complexity of the algorithm is linear, the implementation should behave similar. A similar argumentation holds for memory usage and the number of exchanged messages.

### 6.3 Usability

The *Usability* factor evaluates the effort needed for training people to use the considered software component, to prepare arbitrary input for processing, and to understand the produced output. We will produce documents that describe each of the previous issues to ensure usability of our product. The user’s guide must describe each possible parameter that can be given to the application. Such a document must describe how to parameterize the application and the semantics of each configuration parameter. Also, it must be very clear how to start the application and examples, i.e. configuration files, should be provided. A reader must be able to produce a completely valid configuration and to interpret the output after reading this document.

### 6.4 Maintainability

The *Maintainability* factor evaluates the ease of effort for locating and fixing a software failure. The evaluation of maintainability of a particular software component will be done through TRAC and Hudson (see Chapter 5). We use two levels for maintainability: low and high. If a project (i) adheres to the naming standards, (ii) follows the common directory structure, (iii) the source is documented using doxygen and (iv) the component provides complete build scripts, we say that the component has
high maintainability. If one of these four criteria is not matched, we say that the component has low maintainability.

6.5 Verifiability

The Verifiability factor evaluates the effort to verify the specified software operation and performance. Here, we do not consider verifiability of the algorithm itself; this is considered during the design of the algorithm and in FRONTS this is usually done through rigorously analyzing the performance of the algorithm using mathematical tools. We wish to evaluate the effort for verifying that the implementation follows closely the original algorithm and that the heuristics introduced indeed improve performance. Note however that the verification of distributed algorithm is a tedious task even when the number of nodes is less than 10.

For these reasons, in order to increase verifiability, we adopt the following guidelines for component developers:

- Implementation must include debugging messages when the component sends or receives data.
- Implementation must include debugging messages when the component changes state.
- Code must adhere to coding styles as defined in Chapter 4.
- Code must provide detailed and meaningful comments, at least at a ration 2:1 (lines of code : lines of comments).

6.6 Re-usability

The Re-usability factor evaluates the effort to convert a software component for use in another application. For components implemented only at version A, i.e., using group specific software/hardware, the corresponding re-usability will be low. Essentially the components can be used only in applications developed by the same group. For components implemented at version B, C, D or E, re-usability is high since in Shawn automatically supports the interconnection of components.

6.7 Interoperability

The Interoperability factor evaluates the effort to couple the software to the software of another system. For components implemented only at version A, i.e., using group specific software/hardware, the corresponding interoperability will be low. Essentially the components can potentially interoperate with components by the same group. For components implemented at version B, C, D or E, interoperability is high since in Shawn automatically supports the interconnection of components.
6.8 Expandability

The Expandability factor evaluates the effort to enhance current or add new functionality. To ensure this, the software must be based on well-known concepts and libraries. Hence, software in FRONTS must be implemented using the C++ language as defined by the ISO/IEC 14882:1998 standard (sometimes referred to as C++-98) without any non-standard extensions. If software is implemented for Shawn only, code should make use of C++’s Standard Template Library as defined by the standard. The use of additional libraries is limited to Shawn and iSense. Hence, code must only include and use STL, Shawn and iSense to comply with our definition of expandability. However, in some cases, it may be necessary to define exceptions to this rule for specific software components. Exceptions are only be granted by decision of the FRONTS Technical Committee.
References


