4

Automated Testing and Development of WSN Applications

Mohammad Al Saad, Jochen Schiller and Elfriede Fehr
Freie Universität Berlin
Germany

1. Introduction

Over the course of time the application range of Wireless Sensor Networks will become more varied and complex. A WSN may consist of several hundred sensor nodes, which are independent processing units equipped with various sensors and which communicate wirelessly. WSNs can be compared to wireless ad-hoc networks, but the sensor nodes are constrained by very limited resources and suit the purpose of collecting and processing sensory data.
Therefore it is increasingly important to programme it with the corresponding efficiency. Programming can become more productive and robust, if it is subject to a systematic and structured software development process, which enhances application and accommodates for the sensor network's operating conditions. The pivotal approach for this can be found in the automated Software development process, during which administrational functionalities, which are suitable for the operation of the Sensor Network, are integrated. This constitutes the approach of our proposed Tool-Chain ScatterClipse (Al Saad et al., 2008b). The architecture centric method of the model driven paradigm (Stahl et al., 2006) is used for the automation. New in this case is that the models are not only used for documentation or visualisation: The semantic and expressive formal models also act as a method to completely and concisely represent important concepts as well as the domain's (platform's) basic conditions. Such specific, yet technology neutral models, are inputted into the configurable code generator and after their validation the corresponding software artefact is generated and distributed to the appropriate platform (wireless sensors nodes).
The high degree of automation accelerates the development and testing of applications, which are already running on sensor nodes. Furthermore substitutability and reusability of the software artefacts are increased, because the artefacts, alongside the automated code generation, are represented by their respective models. Both increase the development process's productivity. The model driven code generation is used to furthermore generate a largely tailor made code, so that only the required amount of code is generated for the sensor node's intended roll. Thus the scarce memory space is not only optimised, but also unnecessary calculating and energy intensive software modules are avoided. The decreased portion of manually written code also reduces the possibility of a programmer's careless mistakes. In this process the validation on the model level plays an important role, because

the earlier a mistake (bug) is discovered in the development process the more robust and reliant it will become.
For automation purposes an appropriate generative infrastructure was developed ScatterFactory (Al Saad et al., 2007a) and ScatterUnit (Al Saad et al., 2008a), which constitutes the backbone of our platform or Tool-Chain. For the modelling a graphical editor, based on the Eclipse Modelling Framework and the Graphical Modelling Framework (GMF), was developed. For the examination of the basic conditions, which are linked to the respective models, a real time validation was integrated into the editor, which also makes the development process more robust. OpenArchitectureWare (oAW) framework is used as the code generator, where the corresponding code is automatically generated from the inputted model and this code is then deployed onto the deployed sensor nodes. All frameworks are Eclipse platform open source projects.
Furthermore the emphasis lies on the integration of essential functionalities, which regard the administration and Management of the Wireless Sensor Network, with the model driven software development process. These shall not be isolated, but shall be seamlessly combined with attributes like configuration, bug fixing, monitoring, user interaction, over the air software updates as well as sensor status visualization (Al Saad et al., 2007b). This combination potential is an important character of the platform. The realisation of such combinations was achieved by the plug-in oriented architecture in accordance with the Eclipse platform. On the one hand the user can operate certain plug-ins (functionalities) independent from each other, so that a "separation of concerns" is achieved, and on the other hand the user can navigate the different plug-ins collaboratively at the same time, whereby coherence is achieved. In order to improve the platforms productivity, its main features can be accessed in local as well as in remote, or internet based, mode. For this reason one can, for example, operate the administration and configuration from a computer in one location (for instance in a development or test laboratory) while the sensors are deployed in real world conditions (for example an experiment field) in a different remote location. This was realized by an ordinary client/server architecture.

1.1 ScatterWeb WSN-Platform

ScatterWeb (Schiller et al., 2005) is a platform for teaching and prototyping WSN, which was developed by our Work Group Computer Systems and Telematics of the Free University Berlin. The hardware components of the ScatterWeb platform mainly consist of Embedded sensor boards (ESBs), the newly developed configurable Modular sensor boards (MSBs) and and the sink (eGate), which is connected to the PC via USB (see Figure 1). The sensor boards have in addition to a controller and transceiver many functions at its disposal, such as a sensor for luminosity, vibration, temperature and IR movement detection, a beeper, LEDS (red, yellow and green), as well as a microphone. Thus a prototype of a comprehensive monitoring sensor is created, which makes studying the insertion of WSNs in various areas and scenarios – like environmental monitoring, intelligent buildings, Ad hoc process control, etc. – possible. With this ability, various applications running on the computer can communicate with ScatterWeb sensor boards via the eGate, and vice versa, which makes data-gathering, debugging, monitoring, over the air software updates, etc. possible.

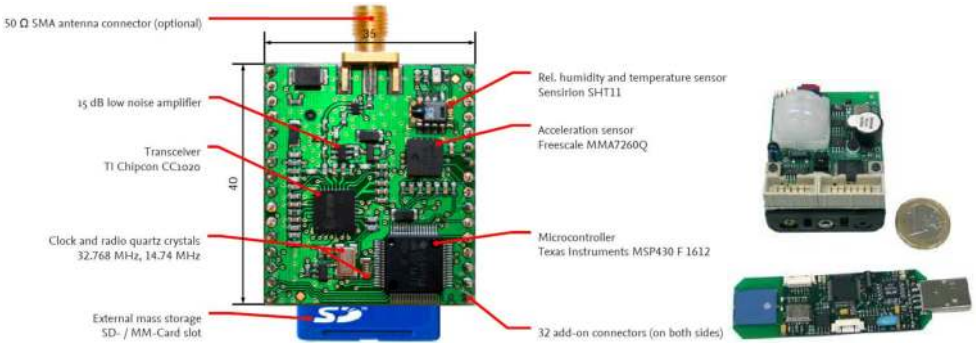


Fig. 1. ScatterWeb WSN-Platform: MSB left, ESB top right, eGate down right

1.2. Architecture Centric Model Driven Software Development (AC-MDSD)

While the main objectives of the OMG relative to the Model Driven Architecture (MDA) are the increase of the portability and interoperational ability of the software on a universal basis, the architecture centric model driven software development (AC-MDSD), as the name states, puts the focus on each an application domain. Instead of generating the same software for different platforms, the AC-MDSD has the goal of variations of software (software families) for a certain domain to automate as much as possible. This attempt is motivated with the observation that the (self repeating) infrastructure code has a considerable part of the entire code-basis in similar applications. With eBusiness applications, it lies around 70%, but with programming closer to the hardware, for instance with embedded systems, this share lies often between 90 and 100% (Eisenecker & Czarnecki., 2000). Consequently it is naturally preferred to create the part automatically so that the actual application specific logic can be concentrated on. In this way, the concentration is set on an application domain for a model language, which would allow the concepts for the underlying platform to be domain related and precisely expressed.

Such a domain specific language (DSL) has as advantage over the usually more complex UML-based models used in the MDA, that the models created in it have a more complete knowledge of the domain. Since the model elements of DSL stand for concrete architectural concepts or aspects of the domain, a model written in DSL offers a higher abstraction level, but is concrete at the same time. The semantic gap between model and code becomes smaller. As a side-effect this simplifies the transformation of the models to code, because the step-by-step refinement of the models to code can often be skipped, since the underlying platform is known and clearly restricted. Overall, the objective target of the paradigm of the AC-MDSD can be compared to the use of modern product lines in the automobile industry.

At the beginning stands the prototype (Reference Implementation), in which the most important concepts are included. The prototype shows what the vehicle that is to be produced is supposed to look like. The construction plans (Models) serve as the starting basis for the end product (Generated Artifact) and point out which units (Components) are required.

In order to simplify the construction of the product line (Generative Architecture), as well as the later production (Code-Generation), logical coherent Components are summarized to production units. Production units, which are not automated or are too complicated to

automate, have to be done by hand (Manual Code). To offer a wide production palette (System Family), the components, as a rule, have to be varied during the production process, while the production platform as such is left unchanged. Thus in context of the AC-MDSD, this approach is also called Product Line Engineering (see Figure 2).

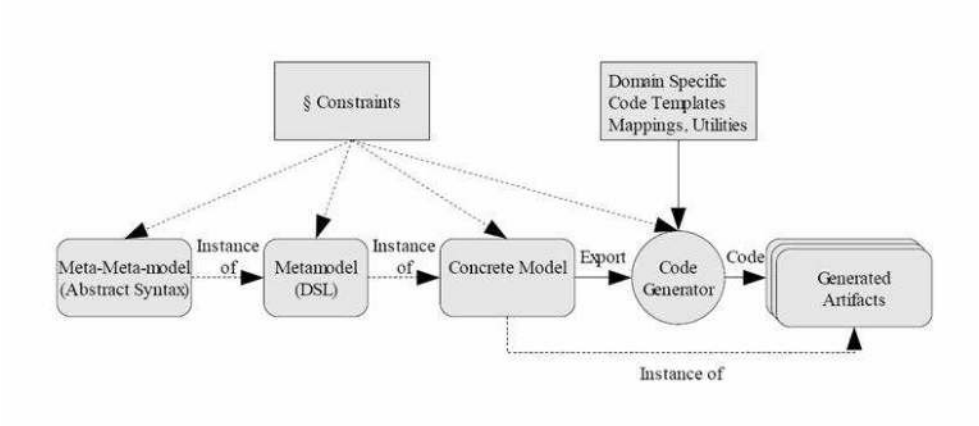


Fig. 2. AC-MDSD as product-line

2. Testing Track

In our research, we examined how tools can support the person who writes test cases. With this in mind, we particularly looked at automated testing of applications for wireless sensor networks (WSNs). A WSN may consist of several hundred sensor nodes, which are independent processing units equipped with various sensors and which communicate wirelessly. WSNs can be compared to wireless ad hoc networks, but the sensor nodes are constrained by very limited resources and suit the purpose of collecting and processing sensory data. To test WSN applications, many common services are needed, e.g. the simulation of sensor input. To provide those services, we implemented a testing framework for our WSN platform ScatterWeb, called ScatterUnit.

2.1 ScatterUnit

Our aim was to create a general-purpose testing frame-work that enables automated tests of WSN applications in respect to component, integration, and system tests. At first, we looked at the spectrum of WSN applications that will potentially be tested using our framework in order to elicit the requirements. A typical WSN application is to collect sensory data over a period of time, which is then evaluated on a PC connected to the WSN, e.g. to keep an eye on water pollution in a river (Akyildiz et al., 2002). A test scenario we may want to apply works as follows:

1. Five sensor nodes to form the WSN.
2. All sensor nodes collect sensory data.
3. The collected data is read out by a PC connected to the WSN.

To write an automated test case for this test scenario, we need a testing framework which is able to simulate sensor input, to invoke the functionality of the WSN application to read out

the collected data, and to observe if the correct data is transmitted to the PC. Thus, the testing framework has to orchestrate the WSN application with mechanisms that in general enable the test case to control the actions and observe the behaviour of the application. When orchestrating the application with these mechanisms, we have to mind the intrusion effect (Cunha et al., 2001): Because the mechanisms allocate resources on the sensor nodes (e.g. processing time) they inevitably influence the execution of the application. This intrusion may cause the application to fail, which it would not have without the orchestration. This may be the case for applications which must react quickly and are orchestrated with mechanisms that allocate significant processing time. Thus, the mechanisms must not allocate resources that influence the execution of the application unfavourably. The mechanisms needed to test the WSN application we mentioned above allocate mainly processing time. This does not lead to an unfavourable intrusion since the application is not constrained by timeliness. Because our aim was to create a general-purpose testing framework, we do not consider WSN applications with stringent timeliness constraints.

However, we found an unfavourable influence the orchestration can have on the execution of the WSN application being tested when we looked at another typical test object: We may also want to test a service used by many WSN applications, such as a routing protocol. A simple test scenario is to establish a small multi-hop network and to send a data packet from one sensor node to another, whereas the routing protocol has to forward the data packet on one or more intermediate nodes. A test case for this scenario sends a data packet by using the functionality of the routing protocol, observes through which intermediate nodes the data packet is forwarded, and asserts that the data packet is received by the destination node. When we execute this test case, the routing protocol may fail to deliver the data packet because it was forwarded on a wrong route. To understand at which point the routing protocol actually failed, we have to reconstruct the route the data packet took. For that, we need the information of the observed forwarding actions on the intermediate nodes in the correct order. Because this information is retrieved on different sensor nodes we have to gather it at a central place for evaluation. A testing infrastructure that is able to do so is SeNeTs (Blumenthal et al., 2004): A base station sends commands to the nodes in the network – e.g. to initiate sending the data packet – and the nodes send information on all relevant events back to the base station, i.e. where the route the data packet took is reconstructed. However, this architecture was designed for wireless ad-hoc networks where resources are not as limited as in WSNs. For WSNs we noticed that the transfer of a data packet may fail if the radio channel was currently occupied by another sensor node (the reason may be the occurrence of too many collisions on the radio channel). Therefore, we cannot afford to establish a resource demanding communication between a base station and the sensor nodes as SeNeTs does. To avoid an unfavourable influence on the execution of the WSN application being tested, our testing frame-work must not use the radio channel too frequently. Thus, we decided not to use a centralized but a decentralized approach (Rafiq & Cacciar, 2003) for our testing framework ScatterUnit which meets the requirement to produce the least possible intrusion effect.

To avoid sending commands over the radio channel each sensor node is configured with its own set of actions before the execution of a test case is started. Those actions may call a method of the application being tested, e.g. to send a data packet using the routing protocol. They may also simulate an event, e.g. to simulate sensory data input. And they may be used

to start waiting for a specific event, e.g. to wait for the reception of a data packet on the radio channel. All actions which will be executed on the same sensor node are implemented by a node script which also knows when to execute them. Thus, a node script has the responsibility to control the execution of the test case locally on its sensor node. To coordinate the actions executed on different sensor nodes, a command service is provided by ScatterUnit (Ulrich et al., 1999). Sending commands for coordination is the only reason to use the radio channel for testing purposes while the test case is running. During the execution of the test case, all relevant events are logged on the sensor node where they occur. Not until the execution of the test case was terminated the radio channel is used to send the logs of all sensor nodes to a PC connected to the WSN. The PC uses these logs to evaluate the behaviour of the application being tested and to decide whether a failure occurred or not.

How to implement a test case using ScatterUnit is well demonstrated by a test case to test a routing protocol. Especially in the field of WSNs, routing protocols must have the ability to adapt to changing network topologies. To test this feature we apply the following test scenario:

1. A WSN is set up with the topology shown in Figure 3 (left). (ScatterUnit provides a topology simulation service which filters out received data packets from nodes virtually out of range.)
2. Sensor node 1 sends a data packet to sensor node 4.
3. Sensor node 4 receives the data packet.
4. The WSN changes its topology in a way that the path between sensor node 1 and 4 changes: Sensor node 4 moves out of range of sensor node 3 and into range of sensor node 2 (see Figure 3 right).
5. Sensor node 1 sends a second data packet to sensor node 4.
6. Sensor node 4 receives the data packet.

Due to the changing topology of the WSN, the routing protocol has to choose a different path to redirect the second data packet. If sensor node 4 does not receive the second packet we would have shown that the routing protocol failed to adapt to the changed network topology.

This test case is implemented by four node scripts – one for each sensor node. ScatterUnit calls a set-up method of those node scripts before the execution of the test case is started. This gives us the chance to initialize the topology simulation service provided by ScatterUnit as depicted in Figure 3.

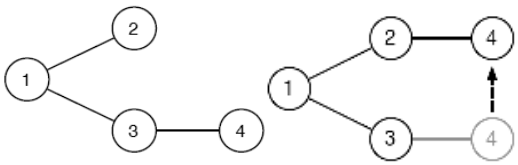


Fig. 3. WSN with four sensor nodes (left: nodes are within communication range of each other, right: simulative modification of the topology while running the test)

This is all we have to implement for the node scripts of sensor nodes 2 and 3. For the node script of sensor node 1, we additionally implement the following: In the start method, which

is called by ScatterUnit once the execution of the test case is started, we prepare a data packet and send it by calling a method of our routing protocol. After the node script receives a command from the node script of sensor node 4 we send the second data packet. If sending one of the data packets fails, we abort the execution of the test case. For the node script of sensor node 4, we implement the following: In the start method we start waiting for the first data packet by using the waiting service provided by ScatterUnit. Once the data packet is received, we reconfigure the topology simulation service by virtually moving sensor node 4 out of range of sensor node 3 and into range of sensor node 2. After that, we send a command to the node script of sensor node 1 to let it send the second data packet and start waiting for it. Once the second data packet is received, we terminate the execution of the test case.

The logs of all sensor nodes which are accumulated during the execution of the test case are sent to a PC and merged into a single time ordered log which looks like this in case the routing protocol failed to send the second data packet:

- The execution of the test case is started.
- Sensor node 4 starts waiting.
- Sensor node 4 received the awaited data packet.
- Sensor node 4 leaves the range of sensor node 3.
- Sensor node 4 enters the range of sensor node 2.
- Sensor node 4 starts waiting.
- Sensor node 1 aborts the execution of the test case.

This log is analyzed by several routines which each check for a certain failure. One of them will report that sensor node 1 failed to send the second data packet. The evaluation of the test results is discussed later in more details.

The outline to implement a test case is a test scenario like the one we enumerated in six steps in the previous section. A test scenario consists of actions – like sending a data packet – and events that are expected to occur if the application being tested does not fail – like receiving a data packet. The first step towards the implementation of the test scenario is to convert each expected event into an action. For example, we have to convert the event of receiving a data packet into an action that starts waiting for the corresponding event to occur. Thus, we have several actions we need to implement in order to get an executable test case. Additionally, we have to specify the order in which these actions are executed. This order is directly given by the test scenario. But to write the code needed to guarantee this order is a complex task.

ScatterUnit requires us to implement a test case by several node scripts. Thus we have to answer two questions by recalling the test scenario:

1. Which actions are executed on the sensor node we want to implement the node script for?
2. After which action has each action to be executed?

If we implemented an action by a node script for one sensor node, we would possibly notice when answering the second question that the preceding action is executed on another sensor node. Actually, this is the case for the action of the test case we introduced in the previous section that sends the second data packet from sensor node 1. This action is executed after the last action to change the topology of the WSN. So, the action is executed on sensor node 1 and its preceding action is executed on sensor node 4. To guarantee the correct execution order of these actions, we have to use the command service provided by ScatterUnit: After the execution of the preceding action is finished, a command is sent to the sensor node

where the other action will be executed once the command is received. The command service used to coordinate the execution of the node scripts is required because of the decentralized approach applied to ScatterUnit. To implement a test case we have to split the test scenario into chunks of consecutive actions which are executed on the same sensor node. We then have to implement all chunks that are associated to the same sensor node by a node script. And these chunks that are distributed throughout the node scripts have to be tied up again by using commands.
To split the test scenario into chunks and tie it up again is a complex task especially for more extensive test scenarios. Obviously, it is not easy to write a test case because the test scenario – which is our outline – cannot be implemented directly. In order to be able to implement a test scenario easily, we have to delegate the task to split and tie the actions. For that, we applied a model-driven approach to ScatterUnit, where we delegate this task to the code generator which generates the node scripts from a test case model whereat the test case model is a direct representation of the test scenario.

2.2 Model-Driven Visual ScatterUnit

Model-Driven Software Development (MDSD) is the field of automated code generation from formal models. A formal model describes a certain aspect of a system in an abstract way. The architecture centric method of the model-driven paradigm is used for the automation. In the context of ScatterUnit, the described system is a test case. Therefore, we model the test case formally and generate the code for the node scripts. The crucial part of the model-driven approach we applied to ScatterUnit is the choice of the aspect of the test case that is modeled in an abstract way: We model a direct representation of the test scenario wherein the corresponding actions, their assignment to a sensor node on which they will be executed, and the order in which they are executed is modeled. Given this information, the code generator can do the job to split the actions into chunks and tie them up by using commands – as we discussed previously – when it generates the code for the node scripts.
When modeling a test case with Model-Driven Visual ScatterUnit (Al Saad et al., 2008a), we start with a diagram like the one shown in Figure 4. It depicts the course of the test scenario we introduced in previously. The notation is very similar to UML Activity Diagrams. It only differs regarding the activities: An activity represents a group of actions which serve a single purpose, e.g. changing the topology of the WSN. Furthermore, the interior of an activity shows which sensor nodes the represented actions are executed on. Thus, the diagram reads as follows: Once the test case is started, two activities are executed in parallel. Sensor node 1 sends the first data packet, and sensor node 4 waits for reception of that packet. After the packet has arrived, the topology of the WSN is changed. Then, the second packet is sent from sensor node 1, while sensor node 4 waits for reception. If the data packet is received, the execution of the test case is terminated.
The purpose of this diagram is to represent the test scenario in an intuitive way. But to be able to generate the node script code from the model, we have to fill in more detail. Therefore, we add an additional diagram for each activity that models the actions that are represented by the activity. Figure 5 shows the diagram that details the activity Change Topology. We have two actions. One for virtually moving sensor node 4 out of range of sensor node 3; and one to enter the range of sensor node 2. These actions are modelled with all information needed to generate the respective calls of the topology simulation service.

Since the actions are inside the box of sensor node 4 – this is how actions are assigned to sensor nodes – the generated code is part of the node script for sensor node 4.

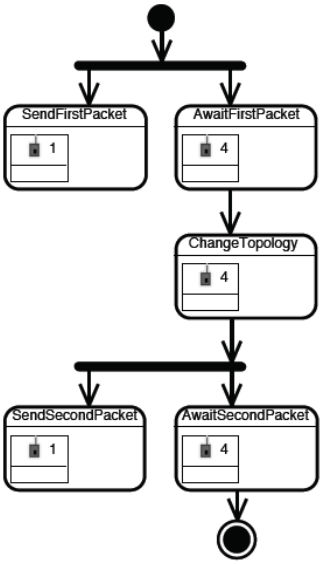


Fig. 4. Test scenario to test a routing protocol

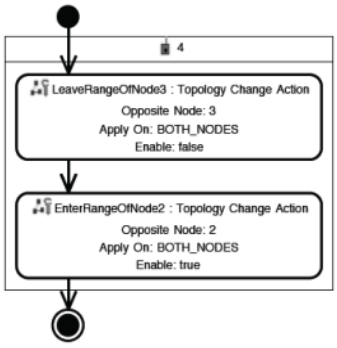


Fig. 5. Diagram that details the activity 'ChangeTopology' in Fig. 4

However, not all the code needed for the node scripts can be generated from the test case model. An action that requires manually written code is part of the diagram shown in Figure 6. This diagram models the actions represented by the activity SendSecondPacket. We actually see just one action with no further detail because the manually written code will do the work. In order to send a data packet over the radio channel using the routing protocol – which is the application being tested – we have to prepare the data packet and call a method of the routing protocol to send the packet. This action is very specific to the application being tested. That is why this action has to be implemented manually. There would be no benefit in trying to model every action in a way that no manually written code

is needed, because the work done by actions varies extensively since we have a wide spectrum of WSN applications potentially being tested using ScatterUnit.

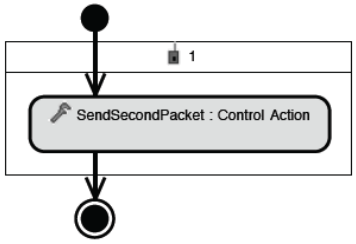


Fig. 6. Diagram that details the activity 'SendSecondPacket' in Fig. 4

Figure 7 shows the diagram which models the actions represented by the activity AwaitSecondPacket: First, the waiting service provided by ScatterUnit is asked to give a notification once the awaited data packet has been received. This notification is represented by the event SecondPacketReceived. Then follows the action AbortIfTimedOut for which we manually implement code to abort the execution of the test case in case the data packet was not received and the waiting job timed out. (Actions which are implemented manually are indicated by a gray background.) If no time out occurred, we actually received the data packet which is logged by the action LogRadioPacket.

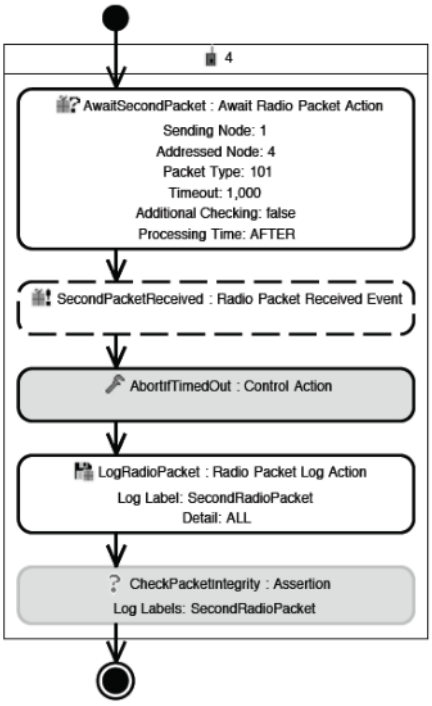


Fig. 7. Diagram that details the activity 'AwaitSecondPacket' in Fig. 4

Referring to this data packet by using the log label SecondRadioPacket, we check its integrity with the assertion CheckPacketIntegrity. Assertions are executed once the execution of the test case is terminated and analyze the log accumulated during the execution in order to check for failures of the application being tested. Since those assertions are application specific, they are manually implemented as well, i.e. we check if the payload of the data packet was not corrupted during transmission. When generating the node scripts from the test case model, the code generator splits up all actions and assigns them to the node script of the sensor node on which they are executed. To maintain the execution order of the actions, the command service provided by ScatterUnit is used. The execution order is modeled through the arrows in the diagrams. Since both actions linked by an arrow are assigned to a sensor node, the code generator is able to decide whether a command is needed to maintain the execution order of both actions, which is the case if the actions are executed on different sensor nodes. For the person who models the test case, it makes no difference if an arrow is drawn between two actions that are executed on the same or on different sensor nodes. Thus, the complex task to write code for coordination purposes is fully hidden from the user which makes modeling the execution order of the actions easy.

The code generated for maintaining the execution order is called infrastructure code because this code is needed in order to write a test case on the ScatterUnit platform. This technical term is used in the context of architecture centric model-driven software development (AC-MDSD, Stahl et al., 2006), which we applied to ScatterUnit: The main purpose of code generation is to generate infrastructure code – for coordination, that is – which is required by the platform, in this case ScatterUnit. As a result, the user can focus on the design of the test scenario rather than having to spend valuable development time on writing infrastructure code which is a complex and time consuming task.

Apart from infrastructure code for coordination purposes, we also enabled the generation of infrastructure code that is needed to evaluate the test results. Once the execution of a test case is terminated, we get a log of all relevant events that have been observed while the test case was running. This log is then analyzed by routines which individually check for a certain failure in order to decide whether the WSN application being tested failed or not. Those routines are represented by assertions in the test case model. For example, the assertion CheckPacketIntegrity shown in Figure 7 represents a routine that checks the payload of the second data packet. If the payload does not contain the expected data, the routine will report the failure that the payload was corrupted.

To run a routine that checks for a specific failure, we have to accomplish two tasks: First, during the execution of the test case, the data must be logged that indicates the absence or the presence of the failure we look at. Second, after the execution of the test case is terminated, the corresponding log entries must be picked out of the log in order to analyze them. We log the needed data by adding log actions to the test case model, i.e. the log action LogRadioPacket. (In contrast to LogRadio-Packet, which needs no manually written code, it is possible to implement application specific log actions manually as well.) To have the corresponding log entries right at hand when implementing a routine to check for a certain failure, log labels are used, i.e. the log action LogRadioPacket declares the log label Second-RadioPacket, which is referenced by the assertion CheckPacketIntegrity. Thus, we do not have to implement code for picking the needed log entries out of the log, because this can be done by the code generator which processes the log labels in the test case model.

Although the code for picking the log entries out can be generated, the routine represented by the assertion has to be implemented manually. The reason is the same as for most actions in the test case model: The routine to check for a certain failure is specific to the application being tested. However, there is one typical failure for which the routine can be generated without the need of manually written code. This type of failure requires the following reasonable assumption about the test case model: The modeled sequence of actions represents the test course that is expected in case the application being tested does not fail. Thus, if the execution of the test case is aborted, we conclude that a failure occurred. For example the action SendSecondPacket shown in Figure 6 may abort the execution of the test case to indicate that the routing protocol failed to send the data packet. A generated routine will report this failure by indicating that the action aborted the execution of the test case. Through aborting the execution of the test case a failure can be reported very easily because no manually written code is needed besides a single call by the action to abort. Since failures that can be reported in this way are of common interest we save significant time to implement code for reporting failures.
Altogether we accumulate the following test results: Failures reported by assertions, a failure reported by aborting the execution of the test case, and the data that has been logged by log actions. To facilitate the reading of these test results, we incorporated them into the diagrams of the test case model. The diagram in Figure 8 incorporates the test results indicating the failure reported by the assertion CheckPacketIntegrity. This failure is shown by the lightening icon in the lower left of the assertion. The tooltip of that icon prints: "The payload of the data packet was corrupted." Additionally, the i icon in the lower left of the log action LogRadioPacket indicates the logged data. The tooltip of that icon prints the data of the data packet. Thus, the test results are easily accessible in the diagrams, because each piece of information is assigned to a corresponding action in the test case model. Without the incorporation of the test results, the user would have to read a textual log, with no reference to the test case model. To understand the information given by the textual log the user would have to embed it into the context of the test scenario herself which is a difficult and time consuming task. The improved readability of the test results also makes it easier and less time consuming to use log actions to get insights on the cause of a reported failure. If we got the test results shown in Figure 8, we would look at the data logged by the log action LogRadioPacket and may notice that the payload was still intact but only the last byte was missing.

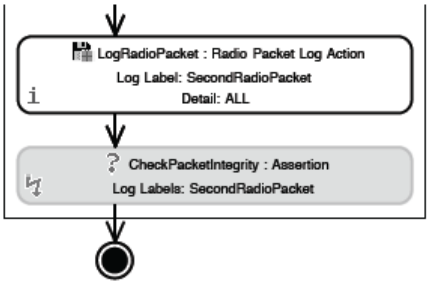


Fig. 8. Test results incorporated into the diagram by adding icons to the lower left of the actions

After we detected a failure by executing a test case, our next step is to fix the fault within the application being tested which caused it to fail. But before we can correct the application code, we have to locate the fault (see Figure 9).

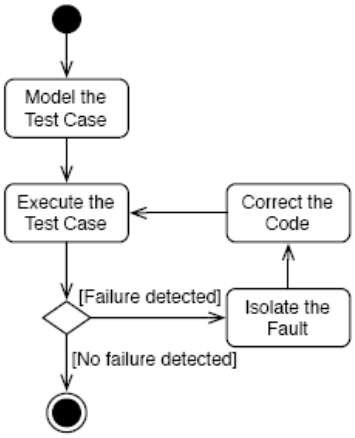


Fig. 9. The process of quality assurance in the context of a test case

In general, we do this by gradually isolating its code location until we can pinpoint the fault. Therefore, Agans recommends using a divide and conquer approach (Agans, 2002). To illustrate this approach, we shall use the test case for the application which measures the water pollution we introduced in Dection 2: Suppose that the test case reports a failure that not all sensory data was transmitted to the base station. The cause may lie within the activity of each sensor node to collect and store the sensory data, or it may lie within the task to transfer the stored data to the base station. To answer which case is true, we next check if all sensory data was stored on each sensor node as expected. If this is true, we have to look for a cause associated to the transmission of the data to the base station. Otherwise (the stored data is corrupted), the code for collecting and storing the sensory data is faulty. Now, we have identified the part of the code where we have to look for a fault. If the application failed to collect and store the sensory data, we may divide the location of the fault again by asking whether the collection task or the storing task went wrong. In general, we iteratively divide the code part where the fault is located and thus narrow our focus until we are able to locate the fault itself.

The process of isolating a fault is a very demanding cognitive task where hypotheses about the cause of a failure are suggested and verified iteratively (Xu & Rajlich, 2004). We do so when using the divide and conquer approach by suggesting a hypothesis that will help us – once verified – to divide the code where the fault is located: In the above example, we may have suggested the hypothesis that the cause of the failure lies within the activity to collect and store the sensory data. To verify this hypothesis, we need to gather information on the data that the sensor nodes actually store. This task to gather information on the behavior of the faulty application is mandatory to be able to suggest and verify hypotheses. Actually, information can be gathered by adding log actions to the test case model, which logs the

needed information. Thus, we refine the test case model in order to get more insight on the cause of the failure reproduced by the test case.
In summary, altering a test case – which is done many times when isolating a fault – would involve writing and rewriting infrastructure code which is required by the platform ScatterUnit. This task is completely done by the code generator of Model-Driven Visual ScatterUnit, which makes the fault isolating process more efficient.

2.3 Model Checking

In order to ensure the generated code's and with it the test cases' quality, the test case model is checked with regard to its syntactic and semantic rules in the form of constraints. This happens not only before the node script's test case model is generated, but also during the modeling of the test case and is called for this reason Live-Validation. As the model validation is run with the help of openArchitectureWare Framework and as the visual editor was created with the help of the Graphical Modeling Framework, the components are combined with the help of the GMF2 Adapters to enable Live-Validation. For this purpose the visual editor embeds the GMF2 adapter, which allows access to the openArchitectureWare's model checking engine. This is repeatedly called to validate the test case model, which is momentarily being edited, based on the constraints. Figure 10 shows ScatterClipse's Architecture regarding the Live-Validation.

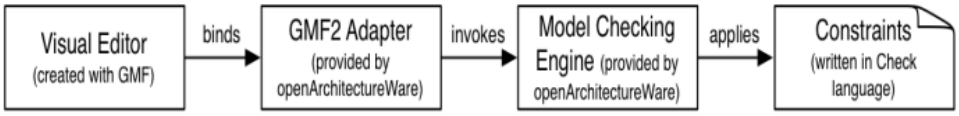


Fig. 10. Live-Validation Components

For instance the command names are used in the generated code's method names, they are not allowed to contain any special characters, so that the code can be compiled (see Fig. 11).

```
context testcase::ControlPath
ERROR "The name must only contain alphabetic and numeric letters." :
this.name.matches("[\\p{Alpha}\\p{Digit}]*");
```

A large number of syntactic rules can be compiled, but the true use of model validation unfurls when checking for semantic rules. For instance as it is defined that a test case model models a course of the test case, it is therefore sensible to check (see Fig. 12), if the test case model consists of a linear sequence of commands:

```
context testcase::ControlPath
       ERROR "A control path must have one incoming link." :
       this.incoming.size == 1;
context testcase::ControlPath
       ERROR "A control path must have at maximum one outgoing
link." : this.outgoing.size <= 1;
```

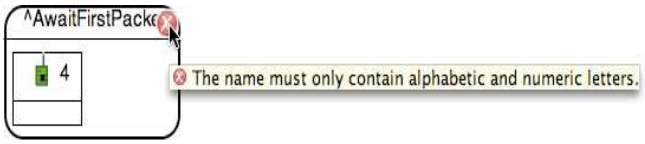


Fig. 11. Live-Validation detected an invalid name

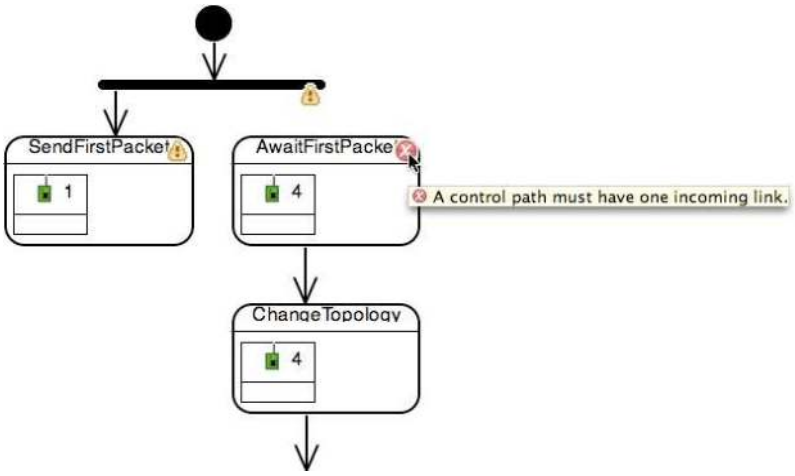


Fig. 12. Live-Validation detected a control path with no predecessor

Overall the model validation secures the quality of the executable test case and increases with it the robustness of the test case development. Furthermore the Live- Validation shows mistakes during the test case modeling, so that the user can correct them in a timely fashion.

3. Development Track

ScatterFactory (Al Saad et al., 2007a) – similar to Visual ScatterUnit - is a generative infrastructure for the model driven development of software for the Embedded sensor boards of the WSN-Platform ScatterWeb. The chosen architecture centric approach represents an instance of the Model Driven Development. The goal is the furthermost automated and standardized production of software system families for the ScatterWeb sensor boards. For this purpose, a component meta model was developed, which builds a basis for a complete tool chain, from the model platform all the way to the deployment of the generated code onto the sensor boards. To model a ScatterWeb network, a domain specific graphical editor was developed on the basis of the Eclipse Modeling Framework and the Graphical Modeling Framework. For the examination of static model constraints, a real time validation was integrated into the editor. The open ArchitectureWare framework was used for the transformation from models into code. The ScatterFactory framework was completed with additional components like assistants or flash-components for the automatic deployment of generated artefacts in an existing network. Our ScatterFactory tool chain was realized with the Eclipse Framework as a basis.

ScatterFactory was originally developed for the first generation ScatterWeb platform – eGate and ESB sensor nodes – and we wanted to keep the advantage of Scatterfactory also for the second generation ScatterWeb platform - MSB sensor nodes. So we developed ScatterFactory2, which was modelled on the principles of the original ScatterFactory. However ScatterFactory2 accommodates now for the innovations and improvements brought on by the second generation ScatterWeb. ScatterFactory was originally developed for the first generation ScatterWeb platform – eGate and ESB sensor nodes – and we wanted to keep the advantage of Scatterfactory also for the second generation ScatterWeb platform - MSB sensor nodes. So we developed ScatterFactory2, which was modelled on the principles of the original ScatterFactory. However ScatterFactory2 accommodates now for the innovations and improvements brought on by the second generation ScatterWeb.
The main difference between the first and second generation ScatterWeb platforms is the new modular design made up of available firmware, system services and hardware drivers, instead of the old monolithic design. Every driver and every algorithmic library has been made available as a library and can be inserted into the run time environment as needed. In this way only the necessary libraries for an application's operation need to be inserted. ScatterWeb's modular design facilitates the configuration of the run time environment enormously and thus lends itself to be represented with a model of an application's run time environment, created of course with the help of ScatterFactory2. Figure 13 shows a model that was drawn with the help of ScatterFactory2's graphical editor.

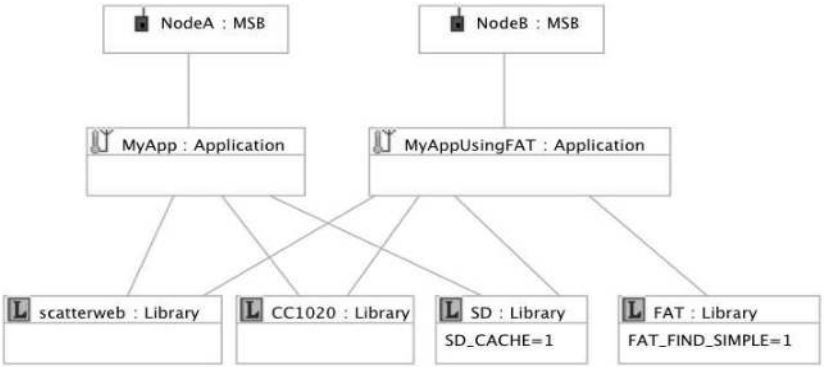


Fig. 13. Model of the run time environment of two applications

The model represents two applications, each with different run time environments: Both applications use the libraries scatterweb and CC1020, which represent the system core. Furthermore the library SD shall also be embedded into their run time environment. This library is a driver, which allows interaction with the sensor node's memory card. However only applications, which allow the management of the memory card's FAT file systems, also receive the library FAT. The diagram elements for the libraries SD and FAT consist, apart from the library's name, of more details, which allow the configuration of the respective library. If the library can be configured with 'Defines' ("#define" of the language "C"), then 'Defines' value can also be placed into the appropriate model. In the case of the SD library, caching for the memory card access has been activated. In the case of the FAT library a search function has been added. These details allow the customization of the individual

libraries to fit the needs of the application. For example the activation or deactivation of the SD library's caching makes a big difference: If the caching is activated, then the data access to the memory card is on average faster. If the caching is deactivated though, then memory space can be saved, which would otherwise be used for the cache and which usually requires about ten percent of all main memory. ScatterFactory2 basically generates out of the model a Make file, which contains the information needed by the translator to insert the libraries as modelled. In order to use the same modelling and code generation tools as in ScatterFactory, the same technologies and tools were used here - especially EMF/GMF and oAW.

3.1 The Integration of Visual ScatterUnit and ScatterFactory2

In the previous examination of the testing process it was assumed that the soon to be tested application already existed and the preceding application development was ignored. An important reason though requires that the application development and the testing process are examined together: An application may need different configurations for different sensor nodes of the same sensor network. Therefore it needs to be taken into consideration with which configuration a sensor node modeled in a test case is associated with. For example it is possible, that not all sensor nodes in a sensor network also have the same sensors on board. A test case therefore, which simulates sensor measurements, may only simulate sensor measurements on sensor nodes which actually have these sensors on board.
As the run time configuration needs to be examined, it lends itself to unite Visual ScatterUnit and ScatterFactory2, because in a sensor network the sensor node's configuration can be configured with the help of ScatterFactory2. The aim during the integration was to associate the sensor node modeled in the test case with the modeled application, which had its run time environment configured with the help of ScatterFactory2 (see Figure 14).

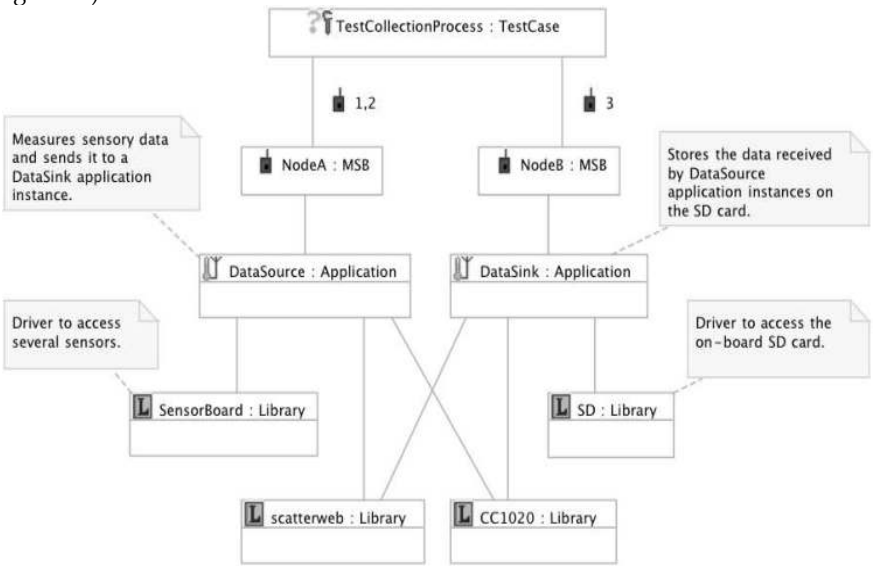


Fig. 14. Test case within an application

In order to make this association the model is supplemented with diagram elements, which represent test cases. The modeled test cases are connected with the modeled sensor nodes. The connection indicates which sensor nodes of the test case are mapped onto which modeled sensor nodes. The modeled test case TestCollectionProcess consists of three node scripts, whereby the node scripts for the sensor nodes 1 and 2 each run on one sensor node, which runs the application instance DataSource and the node script for sensor node 3 runs on one sensor node, which runs the application instance DataSink. If the node script is generated, one receives for each of the three modelled sensor nodes a Make file. Each Make file consists of instructions for the compiler, which makes sure that the correct node script is compiled with the right application instance and the right run time configuration.

4. Management and Monitoring Track

Nowadays apart from text editors, used for editing plain text files, there also exist many different types of editors, for example for the editing of audio and graphical files and of course for web pages. The more complex the edited item is, the higher the demand is for that editor. A good editor should simplify the user's work a lot. This principle can be transferred to WSN, which in the current ubiquitous and pervasive computing era plays a central roll and can be found in more and more areas of application. The challenges regarding programming, monitoring, managing and troubleshooting of WSN increase accordingly and with that the challenges for the corresponding tools as well. The Management Track of the ScatterClipse tool-suit is as an Eclipse based plugin of this manner, which provides the above mentioned services by illustrative means and in so doing, allows the user to utilize them interactively and thus to "edit" the WSN. With the Management plugin's design we attached great importance to the aspect of human-computer interfaces for WSN. Tabfolders, which enables easy user navigation, were used to represent the different features of the Management Plug-in. A supernode can be used as an alternative to the eGate. A supernode is a sensor board with special software, which offers the same functionality as an eGate and which is connected to the computer via serial cable. In this form the supernode can also act as a sink. The Management Plug-in's tabfolder oriented design offers the user the possibility of combining individually required features, so that the collaborating tabfolders function together as a coherent whole. Nevertheless the user can navigate between them at any time, so that a separation of concerns is ensured. The following list represents the Management Plug-in's different features or tabfolders and their functionality:

1. Connection: manages the connection between eGate or SuperNode and the sensor boards.
2. Property: manages the graphical representation of the sensors' status.
3. Terminal: receives and displays information. The user can enter commands in order to configure or control the sensor boards as well as the eGate.
4. Over The Air Flashing: flashes a selected code image Over the Air to the chosen sensor boards and also allows that deployed sensor boards can receive their software updates.

4.1 Connection

Given that the software that runs on the sensor boards is written entirely in the programming language C and given that the Eclipse Framework is written in Java, it has become necessary to develop a bridge between the two systems. This objective is achieved

by a ScatterWeb library for Java that is used to model the ScatterWeb platform into an object oriented paradigm on the basis of the language Java. Thus the communication in both directions between any Java based system and any ScatterWeb WSN, which has been deployed in the real world, is ensured. The connection is made uncomplicatedly by the method eGate.connect() using the javax.comm. Library. The serial port and its corresponding input and output data streams are determined by the parameter names (like Nr. of the COM Port) assigned to the method. Messages are sent to the network, which are necessary for its initialization, and the connection is made.
Figure 15 shows a screenshot of our connection window, through which the connection to the WSN is established, whereupon its components (eGate and all sensor boards) are determined. Firstly the communication type is selected (1). If the local communication type is chosen, the computer connects with the sensors via eGate or supernode. If the remote communication type is chosen, the computer connects as a client with the sensors through the server. When the local communication type is in use the user can determine if communication with the sensors via eGate or supernode is desired (2). In the next step the user selects the COM port's number (3), through which shall be communicated. If this computer acts as a server, then the other clients in the network can also access the sensors. If the local terminal is chosen, then the sensor data will only be shown on the local computer. If the remote terminal is chosen, then the sensor data is shown on the client computers (4). Information regarding the connection to the WSN (like connect time, sensor ID and sensor type) is shown in a table (5). All information of the sensors is stored in the background to memory. The connection is established and disconnected with a mouse click (6) and the WSN can be scanned again (7). Upon starting the Management Plug-in only the connection tabfolder can be seen. The remaining tabfolders are only shown after the connection with a port was successful and after the scanning of the WSN. This contributes to the clarity and eases the interaction with the user.

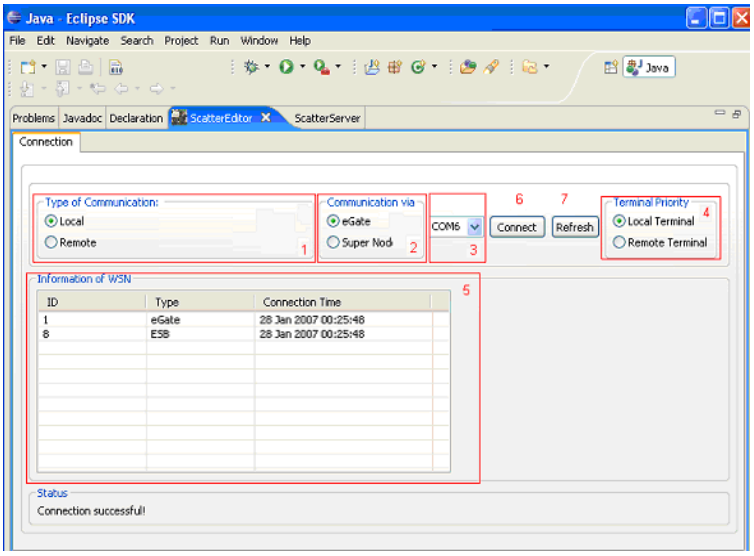


Fig. 15. Connection Tabfolder

4.2. Property

After the connection to the WSN has been established, it is possible to examine the properties of the sensor boards as well as of the eGate. The task of the program section "property" is of graphical representation of the sensor status. Every important sensor state is illustrated "on the fly" and can be configured and controlled through a transparent user interaction. Depending on the sensor's properties, functions can be activated or deactivated, and data, for example the temperature of the sensor's environment, can be shown in this tabfolder. The property tabfolder's visual oriented design is an example for how the aspect of human-computer interfaces for ScatterWeb-WSN is realised. Figure 16 shows a screenshot of property tabfolders. Firstly the user selects a sensor (1). The user updates the list of available sensors by pressing the refresh button and all available sensors are listed in the pull-down menu. In our example we chose the sensor with the ID 8. The IDs can be changed in the field "change sensor ID" (2). If applicable, information concerning the eGate or the supernode is shown (3). The LED's control panel on the sensor (4) can be switched on or off by pressing the appropriate button. The sensor can be restarted (reset) with the restart button, just as the beeper (6). The configuration of the Announce-Flags Serial and Radio (7) as well as the configuration of the Firmware-Flags Programmable and DCO-Checker (8) can be read out and changed. The data from the different measurements (like temperature, volume, movement and vibration) in the sensing field are shown (9). Also shown are the values for Transmit Power and Receive Limit (10), as well as the status of the battery voltage and the optional external power supply (11).

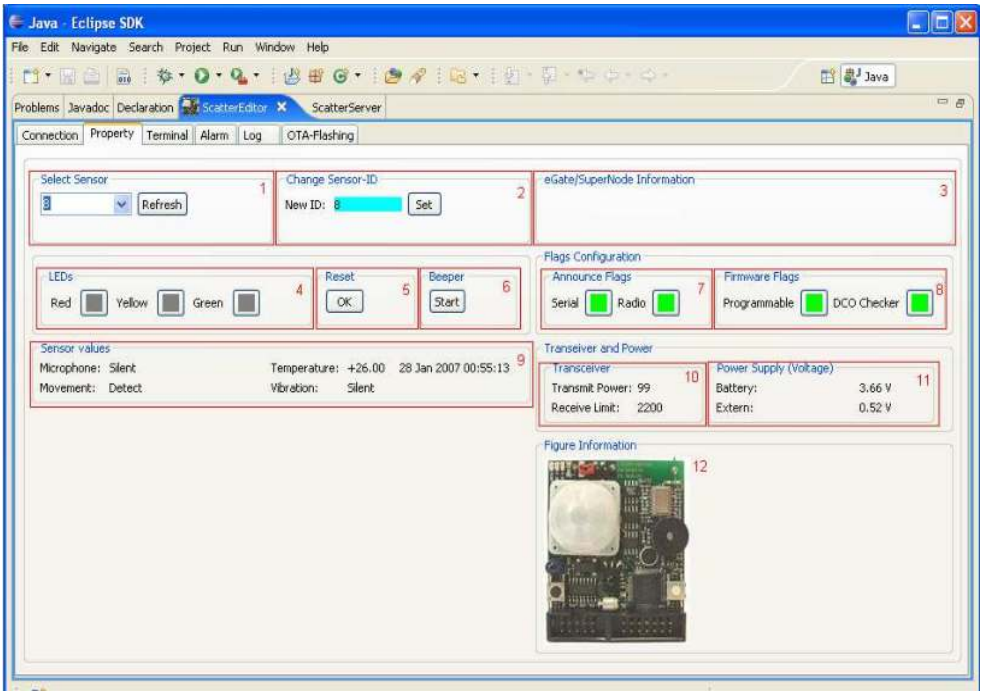


Fig. 16. Property Tabfolder

4.3. Terminal

The terminal offers an easy approach to configure and control the sensor boards through the eGate (see in fig. 17 a screenshot of our Terminal View). This is achieved by the input of terminal commands, which have a specific, but easy, format. In so doing the user can interactively operate the sensor boards. The following example demonstrates how terminal commands look and how they are used: @21 stp 99. The ID of the addressed sensor board always follows the @ character. If the @, and in so being also the ID, is missing, then the command refers to the eGate. After this the instruction follows. stp stands for set transmission power. Certain commands expect parameters like the command in the example above. In this case 99 stands for the transmission power.

If the commands are sent over the eGate, then they are sent in the form of a text with the help of Javax.comm library through the COM port to the eGate. The received string is then parsed by a specific parsing module in the eGate and interpreted. After this the interpreted string is processed by creating a package, which corresponds to the command, and sending it over the Air to the sensor board. The functionality of every terminal command is implemented by a C macro on the sensor's C level. With this one can flexibly define individual terminal commands and have them carried out by the corresponding implementation on the C level. This eases the conducting of experiments, as well as testing and debugging of newly implemented functions. The top output window (1) in Figure 24 shows the response of the queried sensor board. When the first letter of a command is pressed a list of commands appears above the command line (4) with the same starting letter and these commands are taken on in the command line when they are clicked on.

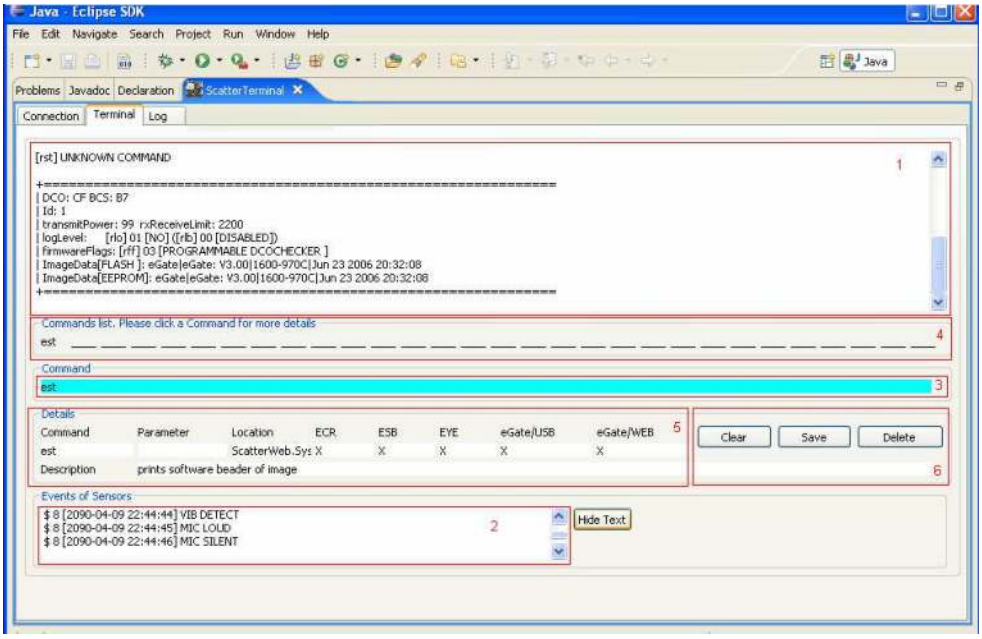


Fig. 17. Terminal Tabfolder

As an assistance the meaning and, where appropriate, the parameters of a clicked on command are displayed below the command line (5).

4.4. Over The Air Flashing

Mass flashing over the eGate is a little more complex when compared to serial flashing. First of all a serial connection to the eGate via Java COM Ports is made. For this task we use the Javax.comn package. Through this the software's binary image (Hex file) is loaded line for line into the eGate's EEPROM. Next the image is sent to all target nodes and at the same time errors, which have occurred, are listened for on the serial connection to the eGate. If necessary this will be announced by the respective dialog box. Another challenge lies in providing the user with the clear and easy interaction possibility with this process, which would improve the reliability and handling of the OTA flashing component in ScatterEditor. Figure 18 shows the GUI for OTA flashing as well as the of interaction with the user after the selection of the Hex file in the relevant folder. As shown in the Connection Tabfolder when the refresh button is pressed, all sensors within range of the eGate are determined and the IDs of the sensors are shown in the corresponding window. Scanning the WSN at the beginning has the effect that only live (non defect) sensor boards come into question for the OTA flashing process. The selected Hex file (1) is loaded into the eGate's EEPROM and the progression of this step is shown to the user by the progress bar (2). During the loading process the eGate sends its respective messages, which are shown in the window (4). When the loading process has been completed, the user can insert the IDs of the senor boards, which should be flashed (3).Thus it is also possible to select several sensors by inserting their IDs and to flash these at the same time with the same Hex file.

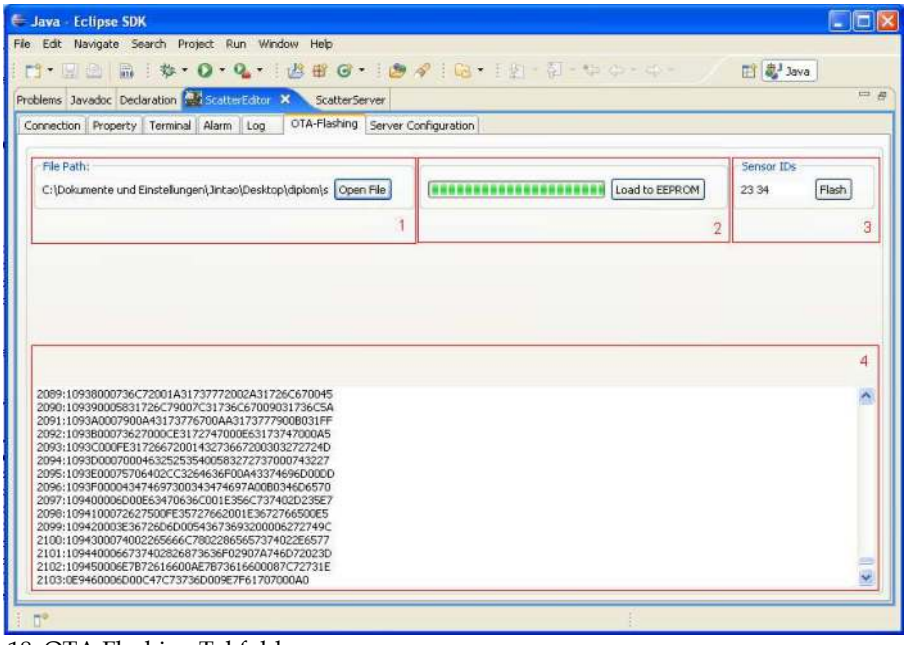


Fig. 18. OTA Flashing Tabfolder

The flashing process begins as soon as the Flash Sensors button is pressed (3). Alternatively one can also flash the eGate by typing "egate".

4.5. Internet Integration

Sensor networks are often deployed in areas, to which people normally have only a limited access, for instance a nature reserve or areas with an extreme climate etc. It stands therefore to reason to connect the management Plug-in with the internet, in order to offer access to Plug-ins's services and features from any, one or more, remote computers (clients). In this case there would not be an eGate connected to these computers (clients). To solve this problem the classical client/server approach was followed by using RMI (Remote Method Invocation, RMI) from Java (see Figure 19). We implemented an eGate server, which runs on the computer, to which the eGate is connected. This computer takes on the role of a server. The Server possesses an interface, which contains the methods, which are available to all clients. Only methods, which are defined on the server, are available to the Client.
The remote function's security is a vital issue. If one wishes to extend the remote option further, then it is important that the client's access rights are clearly defined. That is why the (server) Management Plug-in provides a generated public key during the first program start. The administrator is advised to change the key. Clients can not use old keys to access the server after the key has been changed. The client (user) must be contacted, in order to find out the new key. The procedure, with which the services of the server can be accessed, is, from the point of view of the client, as following: The first step in connecting with the server is the input of the server IP and key. Access to the server will be denied without the correct key. The key, as mentioned afore, can only be changed on the server side. The second security feature is the IP address list, which was set up on the server. From the server this list can be changed, enlarged or deleted at any time.

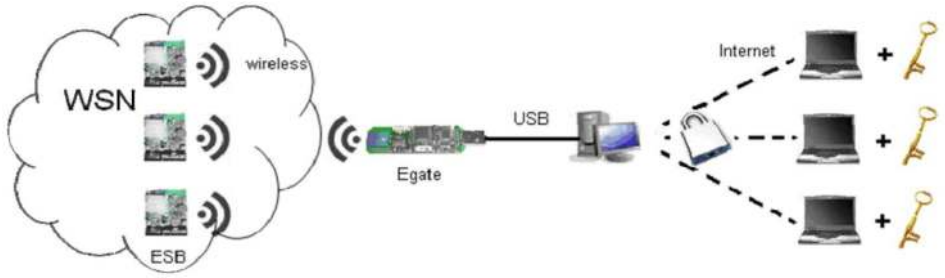


Fig. 19. Server Configuration

5. Conclusions: Putting It All Together

We presented ScatterClipse a model-driven Eclipse-based toolchain for developing, testing and managing Wireless Sensor Networks. The high degree of automation accelerates the development and testing of applications, which are already running on sensor nodes. Furthermore substitutability and reusability of the software artefacts are increased, because the artefacts, alongside the automated code generation, are represented by their respective models. Both increase the development process's productivity. The model driven code

generation is used to furthermore generate a largely tailor made code, so that only the required amount of code is generated for the sensor node's intended roll. Thus the scarce memory space is not only optimized, but also unnecessary calculating and energy intensive software modules are avoided. The decreased portion of manually written code also reduces the possibility of a programmer's careless mistakes. However if the same code had been written manually, then a bug would be more probable. Such a bug results in the test case being defect, which is highly undesirable. This is why the use of a model-driven test environment gives a certain robustness against bugs made during the development of the test case. The model validation also makes a large contribution towards robustness by discovering certain bugs early on, which, if manually implemented, would only be discovered very late in the process.

Hence, several tasks of implementing an a test case and detecting a bug are delegated to the code generator. This is beneficial because these tasks are complex and time-consuming. Rather than performing these tasks himself, the user who tests a WSN application can concentrate on more important matters: Regarding the implementation that is the design of the test scenario; in case of the detection of a bug, that is the decision on an appropriate refinement of the test case to verify a hypothesis. A contribution that could be of general interest is our approach to incorporate the test results into formal models in the context of the Model-Driven paradigm. In so doing models are enhanced with the results of the test process, a method which is practical in other domains as well.

Apart from the interconnection between IP and ScatterClipse, another focus of the tool-suit lies with the aspect of human-computer interfaces for WSN. ScatterClipse provides the opportunity to manage and monitor different characteristics and properties of WSN illustratively and interactively. Since the tool-chain is based on Eclipse, it offers a plug-in oriented architecture and is comprised of free open source components. The tool's plug-in oriented architecture increases the adaptability for the end user and eases the updating of components. Furthermore the open architecture simplifies the tool's expansion, which increases the system's level of interoperability and flexibility.

Thus ScatterClipse is comprised of many tools for the development and operation of WSN applications, which offer a wide spectrum of functionality. This functionality can be used with the help of wizards, editors, views and menu items. However these can only be accessed from many different locations within Eclipse. Hence the user has been missing an overview of the total functionality at his disposal. In order to give the user this overview and a chance to directly access the functionality we developed an Eclipse view called ScatterClipse Assembly Line, which allows the access of all of ScatterClipse's functionality from one central location (see Figure 20).

Development: The Development stage allows access to all of ScatterFactory2's functions. With the first step a wizard is opened, which creates a model file or opens an already existing model file, which then can be used for modelling. The next step activates the code generation. First a project is chosen, in which code should be generated, and then the code generation is activated as pertaining to the model file chosen during the first step.

Testing: The Testing stage allows access to all functions of Visual ScatterUnit and ScatterUnit. Similar to the last stage the first two steps revolve around creating and editing a model file and the generation of test case code. During the third step a portion of test case code (code that will be executed on a sensor node) can be chosen, compiled and then installed onto a sensor node. During the final step, after all portions of test case code have

been installed onto their respective sensor nodes, the test case can be run and the newly created resolution minutes loaded, so that the test results can be visualised in the test case model.

Deployment: The Development stage allows the application's installation and flashing onto the sensor nodes, not for testing purposes, but for actual service. This stage represents the transition between application development and application operation. Furthermore it has been taken into account that applications can be comprised of different parts. These application parts were modelled within the Development stage with the help of ScatterFactory2 and can be individually selected during this stage.

Troubleshooting and Configuration: The Sensor Network can be depicted by using the proper wizards and opening the respective file. Furthermore views supplied by Plugins regarding WSN Management and Monitoring can be opened directly. One is for instance the Terminal-View, which allows Configuration Commands to be sent directly to the sensor nodes.

Resulting from the collaboration between the several frameworks the user can, for example conduct model-driven software testing and development for the sensors with ScatterFactory2 and Visual ScatterUnit, while concurrently he can configure, control and carry out OTA software updates of the sensors with the support of the Management Plug-in and that not only locally but also remotely via the internet.

Overall the ScatterClipse Assembly Line represents the power of the tool chain, gives the user an overview of the available functionalities and facilitates the access of them. Furthermore the ease of access motivates using all tools from ScatterClipse in symbioses. This open architecture also eases the appropriate enhancement of the platform in response to newly arisen questions regarding WSN. A screen-cast of ScatterClipse can be found under (ScatterClipse).

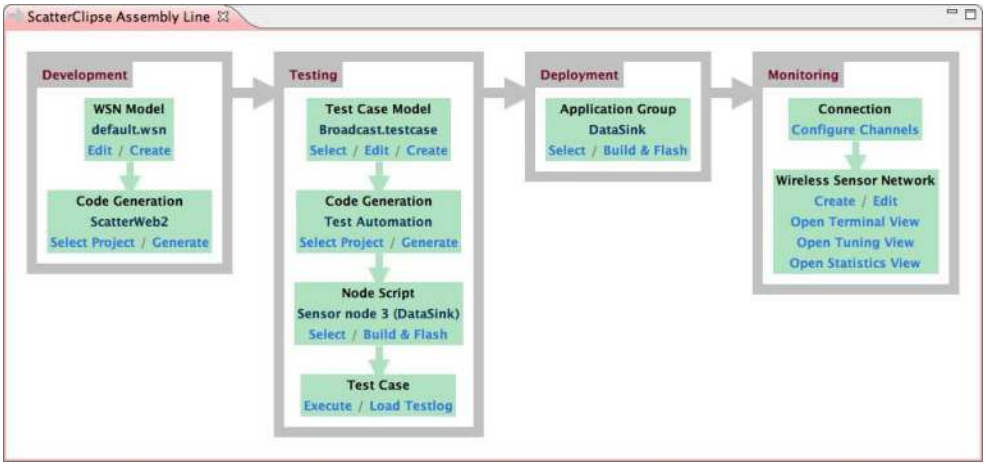


Fig. 20. The ScatterClipse Assembly Line

6. References

Agans, D. J. (2002). Debugging: The 9 Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems, Amacom, ISBN 0-8144-7168-4, New York.

Akyildiz, I. F.; Su, W.; Sankarasubramaniam, Y. & Cayirci, E. (2002). Wireless sensor networks: a Survey. Computer Networks, Vol. 38 No. 4 (2002), pp. 393–422

Al Saad, M., Hentrich, B. & Schiller, J. (2007a). ScatterFactory: An Architecture Centric Framework for Wireless Sensor Networks, Proceedings of the International Conference on New Technologies, Mobility and Security, pp. 12-31, ISBN 978-1-4020-6269-8, May 2007, Paris, Springer, Netherlands

Al Saad, M., Ding, J. & Schiller, J. (2007b). ScatterEditor: An Eclipse Based Tool for Programming, Testing and Managing Wireless Sensor Networks, Proceedings of the International Conference on Sensor Technologies and Applications, pp. 441-450, ISBN 978-0-7695-2988-2, October 2007, Valencia, Spain, IEEE CS Press

Al Saad, M.; Kamenzky, N. & Schiller, J. (2008a). Visual ScatterUnit: A Visual Model Driven Testing Framework of Wireless Sensor Networks Applications, Proceedings of ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems, pp. 751-765, ISBN 978-3-540-87874-2, September/October 2008, Toulouse, France, LNCS Springer

Al Saad, M.; Fehr, E.; Kamenzky, N.; & Schiller, J. (2008b). ScatterClipse: A Model-Driven Tool-Chain for Developing, Testing, and Prototyping Wireless Sensor Networks, Proceedings of 6th IEEE International Symposium on Parallel and Distributed Processing and Applications, pp. 871-885, ISBN 978-0-7695-3471-8, Sydney, Australia, IEEE CS Press

Blumenthal, J.; Handy, M. & Timmermann D. (2004). Senets - test and validation environment for applications in large-scale wireless sensor networks, Proceedings of the 2nd IEEE Int. Con. on Industrial Informatics, pp. 69-73, ISBN 0-7803-8513-6, Berlin, Germany, June 2004, IEEE CS Press

Cunha, J. C.; Loureno J. & Duarte V. (2001). Debugging of parallel and distributed programs, In: Parallel program development for cluster computing: methodology, tools and integrated environments, Cunha, J.C.; Kacsuk, P. & Winter, S. C. (Eds.) pp. 97-129, Nova Science Pub. ISBN 978-1560728658, New York

Eisenecker U. & Czarnecki, K. (2000). Generative Programming. Addison-Wesley Longman, ISBN 978-0201309775, Amsterdam

GMF, http://www.eclipse.org/gmf

oAW, http://www.openarchitectureware.org

Rafiq O. & Cacciari, L. (2003). Coordination algorithm for distributed testing. Journal of Supercomputing, Vol. 24 , No. 2 (February 2003), pp. 203-211, ISSN 0920-8542

ScatterClipse, http://page.mi.fu-berlin.de/saad/ScatterClipse/video.htm

ScatterWeb, http://scatterweb.mi.fu-berlin.de

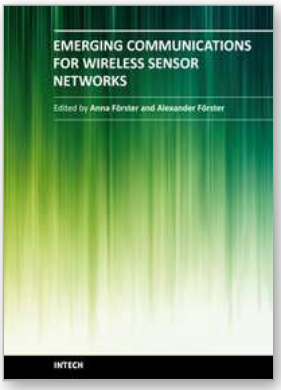
Remote Invocation Method, http://java.sun.com/javase/technologies/core/basic//rmi/index.jsp

Schiller, J.; Liers, A. & H. Ritter (2005). ScatterWeb: A wireless Sensornet Platform for Research and Teaching.Computer Communications, Vol. 28 (2005) pp. 1545-1551

Stahl, T.; Voelter M. & Czarnecki, K. (2006). Model-Driven Software Development: Technology, Engineering, Management, Wiley, ISBN 978-0470025703, Hoboken, New Jersey, USA

Ulrich, A. W.; Zimmerer, P. & Chrobok-Diening G. (1999). Test architectures for testing distributed systems, Proceedings of the 12th Int. Software Quality Week, pp. 24-26, San Jose, California, USA, May 1999

Xu S. & Rajlich, V. (2004). Cognitive process during program debugging, Proceedings of the 3rd IEEE Int. Conference on Cognitive Informatics, pp. 176-182, ISBN 0-7695-2190-8,Victoria, Canada, August 2004, IEEE Computer Society, Washington, DC, USA



Emerging Communications for Wireless Sensor Networks
Edited by

ISBN 978-953-307-082-7
Hard cover, 270 pages
Publisher InTech
Published online 07, February, 2011
Published in print edition February, 2011

Wireless sensor networks are deployed in a rapidly increasing number of arenas, with uses ranging from healthcare monitoring to industrial and environmental safety, as well as new ubiquitous computing devices that are becoming ever more pervasive in our interconnected society. This book presents a range of exciting developments in software communication technologies including some novel applications, such as in high altitude systems, ground heat exchangers and body sensor networks. Authors from leading institutions on four continents present their latest findings in the spirit of exchanging information and stimulating discussion in the WSN community worldwide.

How to reference
In order to correctly reference this scholarly work, feel free to copy and paste the following:

Mohammad Al Saad, Jochen Schiller and Elfriede Fehr (2011). Automated Testing and Development of WSN Applications, Emerging Communications for Wireless Sensor Networks, (Ed.), ISBN: 978-953-307-082-7, InTech, Available from: http://www.intechopen.com/books/emerging-communications-for-wireless-sensor-networks/automated-testing-and-development-of-wsn-applications

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821