Christoph Weyer, Christian Renner, Volker Turau, and Hannes Frey. TinyAID: Automated Instrumentation and Evaluation Support for TinyOS. In Proceedings of the Second International Workshop on Sensor Network Engineering, IWSNE '09, Marina del Rey, CA, USA, June 2009.

TinyAID: Automated Instrumentation and Evaluation Support for TinyOS

Christoph Weyer, Christian Renner, and Volker Turau
Hamburg University of Technology
Institute of Telematics
Schwarzenbergstraße 95
D-21073 Hamburg, Germany
{c.weyer,christian.renner,turau}@tu-harburg.de

Hannes Frey
University of Paderborn
Computer Networks Group
Pohlweg 47-49
D-33098 Paderborn, Germany
hannes.frey@uni-paderborn.de

***Abstract*—TinyAID is a tool that supports automated instrumentation and evaluation of TinyOS-based distributed applications. Two types of instrumentations are provided: logging of call chains and message flows within the network. TinyAID assists the debugging process by post evaluation of the logged data. A main benefit is the visualization component for representing traces in their spatial and temporal order.**

The instrumentation and evaluation concepts are evaluated in two case studies: the SelfWISE framework and a selection of routing algorithms. Due to the automated process of TinyAID the evaluation could be performed without a deeper knowledge of the implementations under test. In the first case TinyAID revealed a weakness in the TOSSIM random number generator. The second case demonstrates the power of TinyAID to visualize the quality of protocols in a unified manner, without any manual changes to the specific source code.

I. Introduction

Sensor networks consist of small, micro controller driven wireless network nodes with additional sensing capabilities. Once deployed in a certain environment, such nodes are supposed to set up a wireless network in an ad-hoc fashion and to run unattendedly for a long period of time. Individual sensor nodes collect data about a certain physical phenomenon. The gathered data can, by means of the wireless network, be easily moved towards one or more data collection nodes.

Sensor networks are a promising approach. In the future they may support measurements in application domains that will be far beyond what can be measured today. The general idea is appealing and attracted many researchers. A lot of research has been conducted on protocol design for various problems, such as data communication, topology control, time synchronization, data collection, tracking, or activity scheduling, just to mention a few. Once a new protocol has been developed and some positive properties have been investigated, theoretically or by means of simulation, producing an executable for real sensor networks still requires a lot of additional effort.

We see that there is a high demand for supplemental development support. One important goal is to support *protocol debugging*, i.e., a common mechanism to test correctness of and to identify errors in existing protocol implementations. Another important goal is to devise a *protocol evaluation* system. Such a system would provide a common base to compare different protocol implementations with each other. This may, e.g., be used to show the benefits and drawbacks of a new implementation as compared to existing protocols. It would also allow for investigating the improvements of a new version over an existing one.

In this work we consider the problem of protocol debugging and protocol evaluation. While at first glance those two problem areas appear to have nothing specific in common, we argue in Sect. II that *automated instrumentation* and *automated evaluation support* is a striking methodology to address both problems. The term automated instrumentation describes the process of automatically adding instrumentation code to an existing implementation, so that the extended code produces output usable for further analysis. The most important kind of output in this context is *logged data*, i.e., producing data sets that systematically encode the protocol behavior and protocol states in a chronological order. These data sets can further be investigated offline. This process is called the *post evaluation pass*. Investigation of the data sets may be performed manually or in an automated way.

From our perspective performing instrumentation automatically requires a framework and coding conventions for the development of the protocols under investigation. Only if the code complies with this, an automatism can find the right hooks to include the appropriate instrumentation code. We believe that TinyOS – having a large community and already consisting of a multitude of relevant protocol implementations – will benefit from automated instrumentation support. In Sect. III we suggest a simple yet effective way for automatic instrumentation of TinyOS-based protocol implementations.

The most relevant elements of evaluation support we see here are *tracing* and *statistical analysis*. Tracing refers to a representation of the behavior of the system under test with respect to temporal or spatial properties. With statistical analysis we refer to all mechanisms that analyze aggregations of logged data. Processing and evaluating large amounts of data is achieved by information visualization that leverages the human visual capabilities, thus allowing to detect complex or even unexpected interrelations. In Sect. IV automatic evaluation examples from both classes are presented.

The documents distributed by this server have been provided by the contributing authors as a means to ensure timely dissemination of scholarly and technical work on a non-commercial basis. Copyright and all rights therein are maintained by the authors or by other copyright holders, not withstanding that they have offered their works here electronically. It is understood that all persons copying this information will adhere to the terms and constraints invoked by each author's copyright. These works may not be reposted without the explicit permission of the copyright holder.

This work is not considering a specific protocol, but introduces a general concept for protocol instrumentation and evaluation. Assessing such a concept is a less common case and some characteristics are hard to quantify. However, we provide partial answers on this difficult question in Sect. V. To test and illustrate usability of the suggested concept, it is applied to existing applications and routing protocols. We also investigate the ease of use of automated instrumentation, the obtainable results from automated evaluation, and the overhead incurred when applying these concepts. In Sect. VI an overview of other existing tools is presented. Finally, Sect. VII will provide a preview on the necessary steps ahead to finally implement our envisioned tool for real deployments.

II. Problem Statement

A. The Need for Debugging Support

Programming usually requires gradually improving a first version towards a final release that offers reliable correctness to a considerable extent. For single-threaded programming the use of breakpoints is the primary choice. However, this becomes ineffective, if applied to multi-threaded or even distributed programming.

In principle it is possible to extend the code for a sensor node with breakpoints, for instance by using the JTAG port of the micro controller. In a deployed system, however, this approach is like removing that node from the sensor network. Reaching a breakpoint disables all interrupts. Hence, the node will not react on any incoming messages or on timeouts. Thus, the inspected system may behave highly different from an uninspected one. Moreover, due to the non-deterministic behavior of system parameters, such as clock drift, message delivery success, or message transmission times, a system may behave differently in each evaluation pass.

We believe that *logging support* is the most appropriate solution here. It is capable of providing a fine-grained chronological list of state information of every node. Once established, the log can be used in a post evaluation pass for further analysis.

However, adding instrumentation code to existing software has a few drawbacks. The additional code changes the runtime behavior. Especially in time critical parts of the source code, e.g., interrupt handlers, the insertion of code can lead to a change in the system behavior. Therefore, the places and the quantity of the instrumentation code must be carefully chosen to keep the influence on runtime behavior as small as possible.

B. The Need for Generic Evaluation Support

When comparing existing protocols with a new protocol implementation, common metrics are needed. Using manual instrumentation for obtaining measurements and computing these metrics causes problems. While adding code lines for logging to an own implementation may be easy, supplementing foreign code with them requires some effort: that implementation has to be studied and understood in the first place. However, necessary places for adding logging code might be overlooked both in the own and particularly in the code of others. As a result, a share of message transmissions will not be available in the message logs, most likely leading to misinterpretation of the protocol characteristics. Having an automated way of instrumenting code will solve this problem. Such a mechanism would add logging code, e.g., at places where messages are transmitted.

C. Limitations

While we propagate automated instrumentation as the primary choice for protocol debugging and comparison, we also see limitations on what can be achieved with that concept. Consider for example an automated message logging feature producing a file containing information about time and visited nodes of certain message instances. Say we want to investigate the success rate of a given single path routing protocol. We can investigate the message type that is used to transmit data with that protocol. Every message instance of this type appears on a certain node, visits a sequence of nodes along the routing path, and finally disappears. However, using such a log file, how can we know that the last visited node was the message destination and that the message was not just dropped due to a routing failure?

We can't. One has to introduce additional coding conventions, e.g., the use of a certain routing framework with generic routing functions for sending a message to the next hop, dropping a message on failure, and consuming the message on success. In this case information about those function calls could be exploited to infer delivery failure and success automatically. However, if we want to deal with legacy code, inventing such additional coding conventions is not an option.

Another important example is a state machine implemented by a large switch statement within the code block of a single module. Depending on the state, a certain part in the switch statement is executed. Since the effects of two states may be exactly the same, despite a local variable encoding the state, it appears to be impossible to infer information of state changes without having some additional knowledge about the code semantics.

It follows that while we can find many application cases where automated instrumentation is a productive tool, there are those cases where only partial information can be produced automatically. Not to exclude those cases, in which additional means for manually adding application context information into the instrumentation and evaluation process are required. For instance, in the routing example discussed above, application context could determine the transport layer end points which were served by the investigated routing protocol.

III. TinyAID Instrumentation

TinyAID currently supports two kinds of automated code instrumentation: *call-chain logging* and *message logging*. Figure 1 depicts the tool chain of automated code instrumentation. Given any nesC source code, the TinyOS tool chain first creates a single, plain C file by combining this code with the TinyOS components used. The automated code instrumentation intercepts the TinyOS tool chain after this point, adding

an additional preprocessing step, called *instrumentation pass*. Given a certain configuration file, "config.cfg" in this example, the *instrumenter* inserts additional instrumentation code, provided by a code template, into the plain C file. The code template reflects the way how the log information is dumped on a given target platform. This step results in an instrumented C file that will then be handed back to the remaining TinyOS tool chain. Depending on the target platform, an instrumented program image or a TOSSIM library, in case of simulations, is finally created.

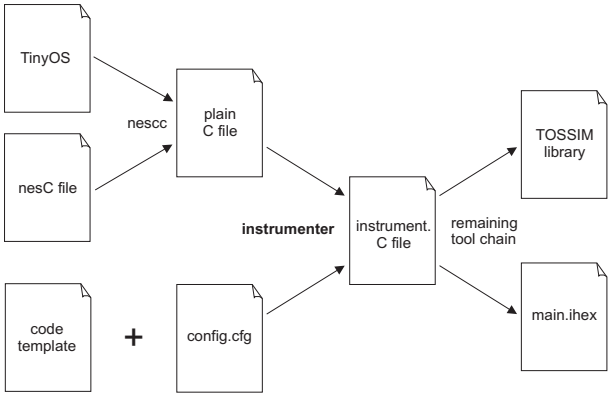


Fig. 1. Automated instrumentation by intercepting the TinyOS tool chain.

A. Call-Chain Logging

In call-chain logging the enter and exit times of certain event handlers and functions of the nodes are logged. This is achieved via additional code that is added to these handlers and functions during the instrumentation pass. For every handler and function, logging code is added immediately after the function entry point, at the end of the function, and immediately before each return. The wiring of TinyOS components is converted into C functions containing only a return statement with the next function call as parameter. These referring functions are skipped by compiler optimizations. Therefore, no instrumentation code is inserted in those functions. Every logged enter and exit event constitutes a single line. Data written in a line is the node ID, a time stamp, either $>$ for enter or $<$ for exit, and an identifier for the called event handler or function.

Since call-chain logging may result in very large data sets, the logged data only consists of unique integers. During the instrumentation pass, a separate file is created, which maps every unique event handler to a unique integer value and vice versa.

An extract of an example call-chain log is depicted in Fig. 2. In this example, event handler 42 of node 5 is entered at time 1320 ms. Within this, a nested call to functions 36 and 12 is performed. Later at time 1684, event handler 20 of node 3 is called, returning immediately without any other nested calls. Finally, at time 1930, event handler 42 is called again, however, now on node 7. This time the handler is passed without any other nested calls.

Node ID	Time [ms]	Direction	Handler ID
5	1320	$>$	42
5	1322	$>$	36
5	1323	$>$	12
5	1324	$<$	12
5	1328	$<$	36
5	1333	$<$	42
3	1648	$>$	20
3	1649	$<$	20
7	1930	$>$	42
7	1931	$<$	42
...	...	...	...

Fig. 2. An extract of an example call-chain log file.

The event or function to be logged is defined by the configuration file used during the instrumentation pass. This information is defined by using regular expressions, which are matched against the unique C preprocessor event handler and function identifications in the plain C file created by the nesC compiler.

Every line in the call-chain configuration file starts with either '+' or '−' to include or respectively exclude event handlers or functions that match the following expression. The inclusion or exclusion symbol is followed by `d`, `f`, or `h` to decide whether the following regular expression is applied on directory names, file names, or handler and function names, respectively. This is followed by the regular expression.

The instrumenter steps through the plain C file and checks for every encountered function or event handler entry point, if they match any of the expressions of the configuration file. For this, the list of expressions is scanned from top to bottom, until the first match is found. Depending on the inclusion and exclusion flag, this line decides, if code instrumentation is applied or not. If no entry is found, the code instrumentation is not applied.

```
-d /opt/tinyos-2.x/.*   # exclude everything in /opt/tinyos-2.x
+f Test.nc              # include everything in file Test.nc
+h fired                # include all fired event handler
+h booted               # include all booted event handler
```

Fig. 3. An example configuration file for call chain logging.

Refer to Fig. 3 for an example. The first line excludes any code residing inside the directory `/opt/tinyos-2.x` from being instrumented for logging. The next line demands that all event handlers and functions implemented in file `Test.nc` are instrumented. The remaining two lines include the event handlers `fired` and `booted` for instrumentation.

B. Message Logging

TinyAID also supports logging of information about messages that have been created, sent, or received by the Active Messaging framework [1]. This is obtained by the automatic instrumentation of the Active Messaging functions. Basically, additional logging code is added immediately after the entry points of the Active Messaging functions `AMSend.send`, `Receive.receive`, and `Packet.clear`. For each supported platform, the configuration file has to provide the

names of the modules implementing these functions. Thus, the instrumenter can automatically insert the code templates for message logging. It also extends the message header by a unique message ID. The latter consists of the address of the node having created the message (the message's origin) and a sequence number. Each message is tagged with this information at creation time. Here we utilize the Active Messaging coding convention that for any newly created message the `Packet.clear` function has to be called. Hence, message tagging is completely transparent to the user.

The actual code for creating the log files has to be provided by the code template file. This file has to contain code snippets which are invoked on message creation, transmission, and reception. In our current example template file, the code snippet logs the following information: address of the message creating node, creation time, character `c` for encoding the message creation event, and message ID (origin plus seqno). For an example, refer to the first data entry in the example log file in Fig. 4.

node	time [ms]	action	type	src	dest	origin	seqno
3	3520	c				3	42
3	3521	s	17	3	12	3	42
5	3524	c				5	14
5	3525	s	34	5	65535	5	14
12	3535	r	17	3	12	3	42
3	3520	c				3	43
...	...	...	...	...	...	...	...

Fig. 4. An extract of an example message log file

For message transmission and reception the corresponding code snippets provided in the code template are also passed. In our current example template, both create a data entry with the same information as provided upon message creation. In contrast, however, send and receive events are distinguished by the characters `s` and `r` in the action column. In addition, the message type, the sender (src) and destination (dest) are also logged. Note that the message ID, consisting of origin and sequence number, can be used to relate packet creation and the origin's final packet transmission, which could be far apart in the log file due to, e.g., message buffering in the routing layer. Furthermore, unicast and broadcast can easily be distinguished, because the latter uses a destination address of 65535. If node and destination address are not equal (and the latter is not the broadcast address), message overhearing can also be tracked.

Refer to Fig. 4 for an example. A message with ID 3-42 is created at time 3520 on node 3. At time 3521 it is then sent to node 12. This node is receiving the message at time 3535. In the mean time another message with ID 5-14 is created at node 5 and broadcasted. Finally, at time 3520, node 3 creates another message 3-43. From the sequence numbering it is clear that this is the next message after the above considered message sent from 3 to 12.

C. Manual Instrumentation

There are two main situations, in which manual code instrumentation may become unavoidable. These include identifying the visited states of certain state machine implementations, and secondly identifying the end points of communication protocols.

For the first aspect, a function `state(name)` is introduced. It can be added manually at any code line. The instrumenter will create a mapping from state names to automatically generated state IDs. Again, as with call-chain logging, the additional mapping is used to keep the logged data compact. Code execution passing such function will produce an entry in the call-chain log file. The direction will be denoted as `!`, and the column handler ID will be used to store the state ID.

For identifying correct delivery of message communication the function `consume(msg)` is introduced, which has to be added at those code places where semantically the message successfully reaches its destination. Whenever code execution passes that function, the message ID and its type are obtained from the given message. In the log data, message consumption is denoted by an `x` in the action column.

IV. TinyAID Evaluation Examples

The automated evaluation process is implemented by the TinyAID *evaluator*. As depicted in Fig. 5, it requires a set of call chain log and message log files in order to produce the appropriate evaluation files. An additional input file `context.dat` provides information about how the files are to be combined, which data has to be extracted, and which additional application context is required to evaluate the data. The output produced can be classified in statistics and visualization data. Statistics data are either data sets or PDF files. Visualization data are either PDF files or videos representing behavior over time.

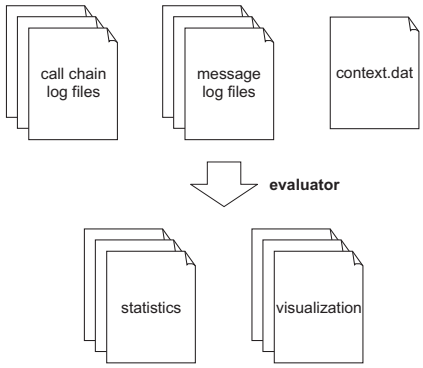


Fig. 5. Automated evaluation based on call chain and message log files.

A. Event Tracing

This concept is a pictorial representation at which time which node was entering a certain event handler. In order to produce such representation, the evaluator requires a single call-chain log file, the information about which event should be inspected for which nodes during which time period, and which resolution should be used for x- and y-axis. The result will be a PDF file with one or several lines of time line

representations. The time is depicted on the x-axis and nodes are depicted on the y-axis. Every time a node enters the inspected event handler, a marker is placed on the time line of that node. Refer to Fig. 6 for an example of a single line event tracing result. Which information can be inferred from this figure?

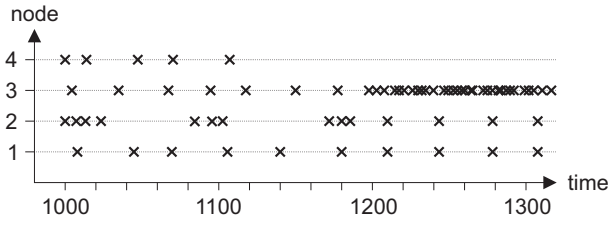


Fig. 6. An event tracing example.

We see several visual indicators which can be useful for protocol debugging. The first one is detection of *event starvation*. This term refers to an event that is expected to occur regularly – e.g., a regular beacon –, but suddenly happens to be missing. Such error might occur in a complex protocol where the scheduled timeouts might be deactivated in certain critical protocol states. If timeout activation on leaving of this critical state was forgotten in the implementation, the timeout may never occur again. For instance, in Fig. 6, starting at about time 1100, node 4 shows an indication of event starvation.

Another indicator that can visually be detected is that of *event explosion*. This term refers to an event that – compared to the event history – starts to occur with an abnormal high frequency at a certain time. Figure 6 depicts an event explosion example. Compared to its event history, at about time 1200, the observed event happens to occur unusually often on node 3. Such an error might occur, when a repeating timeout event handler is supposed to schedule the next timeout whenever it is called. The first time, however, this event has to be scheduled from somewhere outside the event handler code. If by implementation failure, code for scheduling the timeout event from outside the event is accidently executed a second time, two parallel lines of this repeating timeout event will be running on the node. Things are getting worse, if this failure happens to occur from time to time. Eventually the node will exclusively be busy with handling all of these timeout events.

Two other types of visual error indicators refer to the timing of events. One indicator which we call *event jittering* shows a burst behavior of event occurrences. An example can be found for node 2 in Fig. 6. At about time 1000, time 1100, and immediately before time 1200, an event accumulation can be observed while barely any other event occurs in between. This can be an indication of a protocol failure, if regular event inter-arrival times are actually expected. Consider for example the event of sending control information as background traffic. If such burst behavior occurs, control information might severely interfere with actual data transmission or even make data transmission during this burst impossible.

The other visual timing error indicator is *event synchronization*. This refers to a regular event that starts to occur on different nodes at about the same time. Refer to node 1 and 2 in Fig. 6. At about time 1200 both nodes start to enter the event handler almost simultaneously. Such behavior might be an error, if event scheduling on different nodes is expected to be a random process. For example regular beaconing intervals should not be in sync with other nodes in order to avoid too many beacon message collisions. Synchronization might occur, if inclusion of some random component was forgotten in the code or if using an error-prone random number generator for the beacon interval computation.

B. State Tracing

While event tracing aims on a fine grained inspection of a single event, state tracing provides a coarser view on protocol behavior and interaction. The state tracing concept aggregates event handler calls into specific classes. Again for every node a timeline is visualized. The timeline contains bars which extend over the time event handlers from a certain class are called. As soon an event handler from a different class is called, the bar will change. For producing such trace, the evaluator requires the same input as for event tracing. In addition a classification of event handler calls needs to be provided. Such classification declares the classes itself and assigns certain event handler calls to those classes. Any event handler call that is not in this class will be ignored in the evaluation. As with event tracing the result is an PDF file, which contains one or more lines of time line representations. Refer to Fig. 7 for an example of a single line state tracing result.

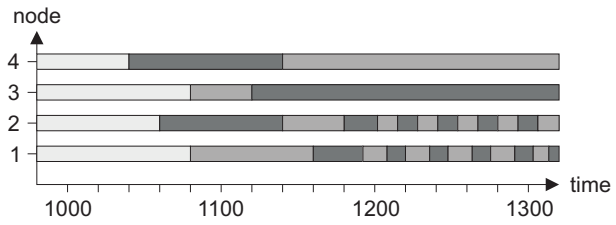


Fig. 7. A state tracing example.

Finding the erroneous code when an error has been found for a certain node at a certain time is one helpful application of this diagram. Consider for example, that a state machine implementation is to be inspected. Say every possible state implements its own handlers for the considered events. Then a reasonable classification will be aggregating the handlers of every state into a different class. Consider now that for the example depicted in Fig. 7 we figured out by some other means that an error occurred on node 3 at time 1200. By inspecting the state trace we know that at that time node was calling event handlers belonging to the dark grey state. Thus, we can narrow our failure search to the event handlers aggregated into the dark grey state.

The diagram can also be useful for detecting the occurrence of *state oscillation*. We denote with this term an unusual frequent change of protocol states on a node. Consider the

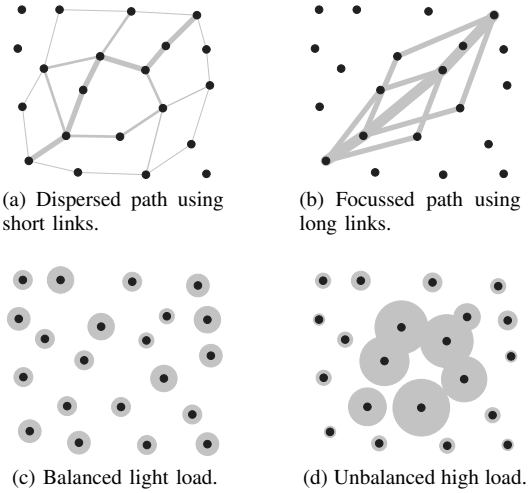
example depicted in Fig. 7. Suppose the colors refer to the states of a clustering protocol. Light grey means undecided, grey means cluster member, and dark grey means cluster head. At 1000 all nodes are in state undecided. Then cluster head and member roles are assigned to the nodes. Around time 1100 roles of 3 and 4 interchange which may be ok if reculstering is an allowed feature. However, at around 1200 nodes 1 and 2 start to exhibit an suspicious behavior with their states starting to change between cluster head and member very frequently. This may be an indication of a failure in the clustering protocols reclustering strategy.

C. Accumulation Diagrams

This visualization feature comprises two classes, which aim at a pictorial representation of network-wide effects at large scale. The first one, referred as *path accumulation diagram* [2], depicts the frequency of links taken by the messages originating from a certain node and belonging to a certain message class (see Fig. 8a and 8b). In our current implementation, the evaluator requires the source node, a single message log file, the type of the messages to be presented, a time frame the figure should be created for, and a mapping of nodes to physical locations in the network.

For each link the number of messages passing that link within a certain time interval is counted. The frequency is then depicted with different line thicknesses, i.e., the most frequently used links are presented with the thickest line. This simple representation is an effective way to get a first impression about the global behavior of a system under test. For instance, routing messages might disperse over the network like depicted in Fig. 8a, disseminating the forwarding burden over the network, but resulting in longer path deviations on the other hand. The opposite behavior, i.e., a protocol that concentrates the path on a few selected next hop nodes, might show a figure like sketched in Fig. 8b. By just visually comparing the diagrams resulting from different protocols, one also gets a first impression about how these protocols may compare in terms of hop count. In the example, Fig. 8a suggests a higher hop count when compared to Fig. 8b. This may also give a first indication on delay characteristics which, however, needs to be investigated separately, if significant evidence is required.

The same visualization principle can also be applied to nodes instead of links in order to show load distribution among network nodes in an intuitive manner. We term this a *load accumulation diagram*. Different node thicknesses are used to visualize load placed on the nodes according to a certain metric. Fig. 8c and 8d show an example load accumulation diagram. Intuitively, Fig. 8c suggests a more balanced load than Fig. 8d, in which a concentration of the load appears in the network center. Assume, e.g., the figures depict the total energy consumption per node during the whole measurement. When aiming at a protocol design that tries to maximize network lifetime by balancing energy load among all nodes, then both diagrams suggest favoring the protocol that produced the left-hand outcome.



(a) Dispersed path using short links.

(b) Focussed path using long links.

(c) Balanced light load.

(d) Unbalanced high load.

Fig. 8. Visualizing large scale network wide effects with accumulation diagrams.

In our implementation the evaluator requires the physical node locations, the evaluation duration, and the evaluation metric. They have to be provided via the context file. Example metrics we considered in this work include the number of message transmissions and receptions, the number of calls of a certain function, and the sum over the nodes sojourn time within a certain state. Depending on the investigated metric, either a single message log file or a call-chain log file has to be provided to the evaluator.

D. Statistics

The concepts presented so far focus on extracting visual data from automatically generated log files. We believe that this is a valuable feature to capture intentioned or failure behavior of a protocol in an intuitive manner. In fact, however, the generated log files offer more than that. Once a significant amount of data has been produced, a rigorous statistical analysis can be run over the data set. We list some common example statistical values that can automatically be extracted from the log files created by instrumented code.

The most common statistical values can be inferred by counting rows or by considering the values of rows in the call chain and message log files matching a certain criterion. One examples would be the fraction of transmissions/receptions of messages of a certain type over the total number of message transmissions. Another one would be using entry time, exit time, and handler ID to infer the number of times a node did enter/exit, and the time fraction a node spent in a certain event handler or in a group of event handlers.

Moreover, we can infer simple statistical data on the spatial distribution of certain message types. Using the message log files' node, origin, and seqno columns, for each individual message we can infer the hop count distance the message travels from its creation until it disappears. Therefore, we are able to compute the expected hop count for each message type. Message types used locally, e.g., for neighbor discovery,

can thus be distinguished automatically from messages used globally, e.g., flooding messages used for route discovery.

We also implemented statistical data extraction about routing paths, which in contrast to the above mentioned requires a bit more than automatically generated log files. Extracting statistics about routing paths require additional context information about the message destination. This is available due to manually inserted `consume` code extensions during the instrumentation process. Such instrumentation will in turn result in message log files that contain information about message reception at the routing destination. An alternative way is to leave the code untouched and provide context information about source and destination nodes during the evaluation pass. Using the message log file's time, origin, and seqno columns, in both cases we are able to infer the traditional routing metrics like average hop count, average delay, and success rate.

V. Concept Evaluation

The concepts introduced so far are evaluated by instrumenting an example application and three routing protocols. In all cases TOSSIM is used for simulating the instrumented code. Here, the process of logging data is simplified by the fact that the information can be logged directly into files. The configuration of the instrumenter for TOSSIM is as follows. We have provided the modules responsible for creating, sending, and receiving packets and code templates producing tracing information as described in Sect. III-B. An example code snippet for tracing packet reception is shown in Listing 1.

In the following a detailed analysis of TinyAID is given. The evaluation of the SelfWISE framework is described in Sect. V-A. The instrumentation for tracing the packet flow in routing protocols is discussed in Sect. V-B and V-C. The overhead introduced by TinyAID when using TOSSIM is benchmarked in Sect. V-D.

```
tossim_header_t  * header = getHeader(msg);
dbg_clear("TINYAID_PACKET_TRACING",
  "%d␣%lld␣R␣%d␣%d␣%d␣%d␣%d\n",
  sim_node(), (sim_time_t)(sim_time() * 1e-7 + 0.5),
  header->type, header->src, header->dest,
  header->origin, header->seqno);
```

Listing 1. Code template for packet reception

Using TinyAID on real hardware in a deployed network requires more advanced mechanisms to collect the logged data. Storing the information on the nodes limits the runtime of the experiment due to the limited memory. The best choice is to have a wired backbone network that can be used for logging purposes. The overhead introduced by logging the data via the serial line is to large (around 2 ms) for most situations. Therefore, a special hardware solution is needed in order to extract the data and collect it at a central point. This additional hardware is part of our future work. In this paper we are concentrating on the principal concept of automated instrumentation and its usability in the area of wireless sensor networks.

A. Use Case: SelfWISE

SelfWISE is a framework for evaluating self-stabilizing algorithms developed at the Institute of Telematics at Hamburg University of Technology [3]. This software contains around 9.000 lines of nesC code. TinyAID is used to inspect the behavior of the SelfWISE framework.

The execution of the self-stabilizing algorithms is based on rounds that must be synchronized and a shared node state that is accomplished by periodical broadcasts. The broadcasts are scheduled at a random point in time to reduce the number of collisions. To evaluate the quality of the implementation, the event inspection of TinyAID is used. Without any knowledge of the internal structure, TinyAID creates instrumentation points in 101 different functions. To investigate the synchronization of the round timer and the broadcast, two function calls where selected: `SyncTimer.fired` and `Paket.send`. The resulting event trace is shown in Fig. 9. The pipe symbol represents the round timer event and the '×' the time point of the broadcast.

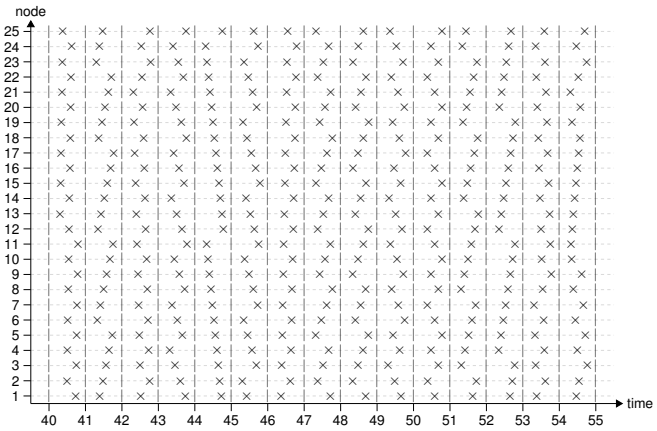


Fig. 9. Event tracing of SelfWISE

This evaluation shows that the synchronization of the round timer works perfectly and that the node state is broadcasted at a random point during the middle of each round. The cyclic patterns of the broadcast were not expected. After further investigations we found out that the implementation of the `RandomMlcgC` component produces insufficient random numbers when using TOSSIM. In real deployments events like packet reception and code runtime introduce a different runtime behavior, so that this effect is not dominant. In TOSSIM, however, the random numbers create these periodical patterns, which leads to an abnormal behavior over time with reoccurring collisions between the same nodes. Changing the implementation of the `RandomMlcgC` components in such a way that it uses the `rand` function of the libc in TOSSIM creates a more randomly distributed message broadcast. Without the automated instrumentation and evaluation

support of TinyAID we may have never become aware of this effect.

The next example evaluation shows the application of state tracing. A self-stabilizing algorithm is executed whenever a node sees an irregularity within its neighborhood. To avoid gratuitous executions the number of nodes that execute the algorithm is reduced by a Bernoulli trial. The behavior over time is shown in Fig. 10. The algorithm execution starts at time 53 and stabilizes at 64. The dark rectangle visualizes that in this round a node has executed the algorithm (done by calling `RuleEngine.abort`) and the gray rectangle indicates that the node wants to execute, but the Bernoulli trial was negative (`RuleEngine_execute`).

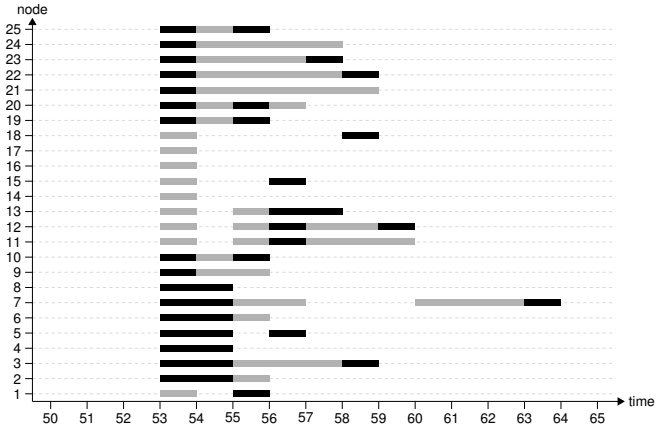


Fig. 10. State tracing of SelfWISE

B. Use Case: Routing Protocols

In this section, we will demonstrate and discuss how our concept of automated packet tracing can be integrated into already implemented protocols. Therefore, three different routing protocols to carry out our analysis on automated packet tracing and the ease of its implementation have been chosen.

The first one is TYMO, an implementation of the well-known DYMO protocol [4], which is included in the TinyOS 2.x code base. TYMO uses internal message types for route requests and route replies. These are sent in order to query and establish a new route, if a forwarding node does not know where to send a given message. The second routing protocol is Dynamic Source Routing (DSR) [5], which follows a similar concept. The third protocol considered is Greedy Routing [6]. Messages are forwarded using the positions of forwarding nodes and the destination. The greedy aspect is realized by each forwarder considering only neighbor nodes closer to the destination and sending the message to the neighbor closest to the destination. The implementations we used for DSR and Greedy Routing originate from [2].

As noted in Sect. III-B, tracing packet sending and receiving is automatically added to the compiled code, if Active Messaging is used. However, `Packet.clear` must be called at the appropriate places in order to make packet tracing work. The following, general issues must be considered first. If the routing protocol under investigation makes use of internal packet types, it must be assured that the corresponding calls to `Packet.clear` are performed, whenever a new internal packet is created. Secondly, routing messages, whether originated on the same node or received and forwarded, must be stored as a complete TinyOS message in a buffer. This is necessary in order to keep message IDs, i.e., packet origin and sequence number, intact.

Calling `Packet.clear` for data messages, that are passed from the application to the routing layer, must be done at the application layer for two reasons. Firstly, following the Active Messaging concept, packet clearing should be performed on creation of a new packet. Clearly, data packets are created at the application layer. Secondly, accurate timing information about packet creation can only be guaranteed, if `Packet.clear` is called upon actual packet creation.

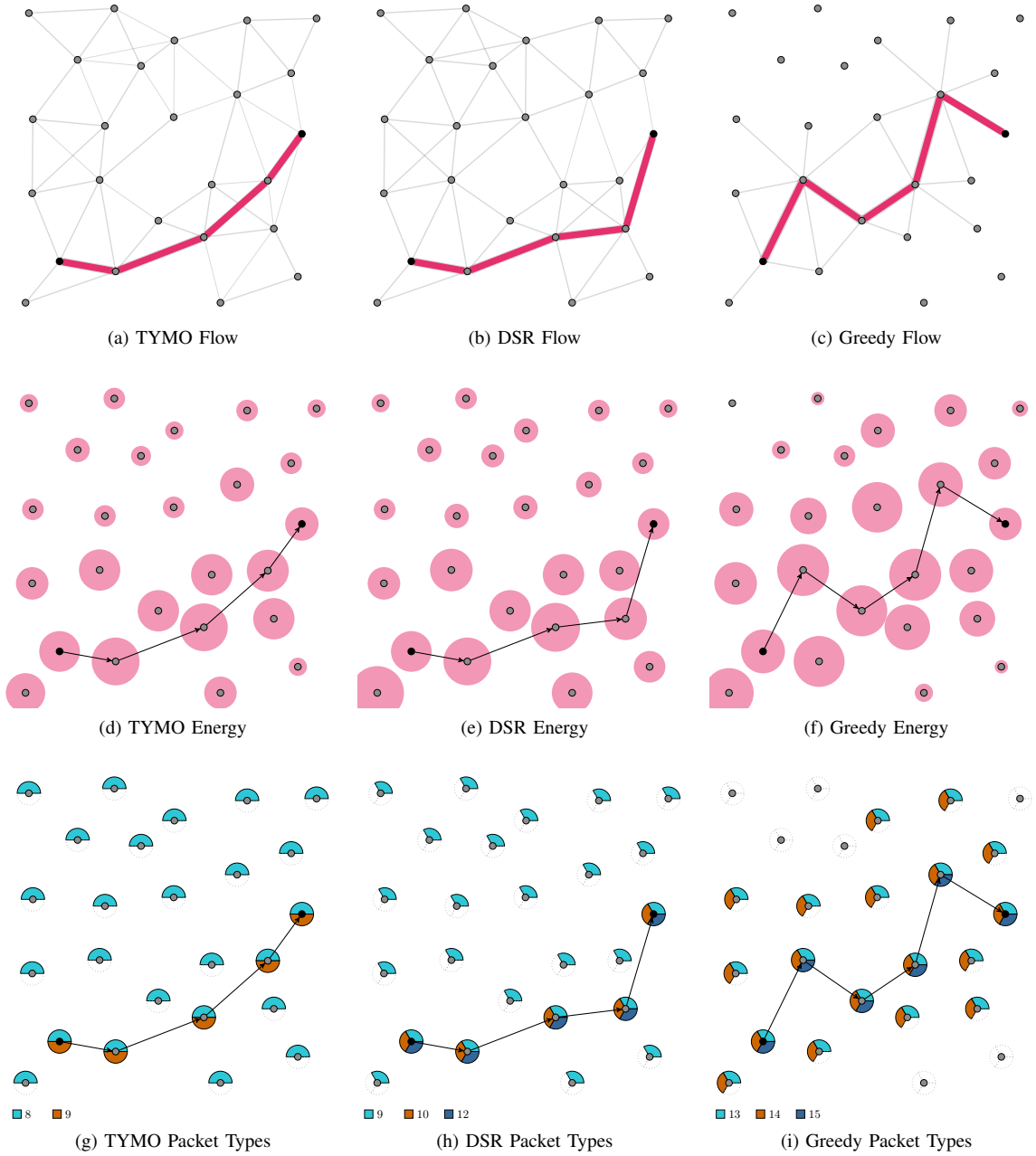
In order to prepare the three routing protocols for automated packet tracing, few changes and additions had to be applied. Neither the implementation of DSR nor Greedy did use the concept of `Packet.clear`. Hence, the corresponding calls had to be added for all internal packet types. Furthermore, both implementations only copied the data part of forwarded data messages into the buffer, although the latter consists of complete TinyOS messages. A minor change in both protocols expunged this problem. These simple and quickly applied changes enabled the two protocol implementations to support automated packet tracing. TYMO does not make use of `Packet.clear` either, so that the appropriate calls had to be added. Moreover, it knows but two packet types. One is intended for internal message exchange, i.e., sending protocol data, whereas the other one is used for routing messages. The former uses subtypes that are stored in the actual packet payload to allow for different protocol message subtypes. As a result, it is required to edit comparably many lines to make the different subtypes visible in the trace log. For a first analysis of our tools, we decided not to take this step and abdicate tracing of the different subtypes.

Besides preparing the routing protocols, we added the required calls of `Packet.clear` upon packet creation to our test application. We also added `consume` upon reception of data messages on the destination node.

C. Performance Metrics

In order to illustrate the power of packet tracing, we will show and discuss visual and statistical analyses obtained from automated traces. We have used the routing protocols introduced in the previous section to show the benefit of packet tracing. The figures and tables in this section have been created with analysis tools that take origin, destination and inspected types as parameters.

All results are based on the same topology of 25 nodes, where the same two nodes serve as data origin and data sink. 10 data packets are created by the origin with a period of 2 seconds. One simulation has been run for each of the three routing protocols. At this point, it is not our intention



(a) TYMO Flow (b) DSR Flow (c) Greedy Flow

(d) TYMO Energy (e) DSR Energy (f) Greedy Energy

(g) TYMO Packet Types (h) DSR Packet Types (i) Greedy Packet Types

Fig. 11. Comparison between different routing protocols: packet flow, energy consumption and packet types

to actually compare the three routing protocols. We plainly are concerned about outlining illustrative examples of utilizing packet tracing for protocol analysis and comparison.

Figure 11 visualizes packet flow and an energy-consumption equivalent. Packet flow, as depicted in the upper row, shows the number of packets intentionally sent on each link, i.e., not counting snooped (overheard) packets. The number of packets sent between two nodes are used as a linear scaling factor for the displayed line width of the corresponding edge. From the visualization it is apparent that TYMO and DSR cause packet flow in the whole network in order to establish routes. In opposition to this, Greedy (Fig. 11) causes local packet flow only, because routing decisions are based upon the position of the destination and a forwarder's neighbors. Packet flow between origin (lower left) and sink (upper right) is considerably higher than in other regions of the network. Furthermore, the share of packet flow caused by routing packets can be compared to that caused by internal routing protocol messages on the actual data paths (gray versus magenta edge widths).

The mid row of Fig. 11 depicts the number of sent and received messages per node. This metric can be used as an energy-consumption equivalent. All figures reveal that energy

	TYMO	DSR	Greedy
Number of Packets			
Created Data Packets	10	10	10
Sent Total	68	68	78
Sent Broadcast	24	24	5
Sent Unicast	44	44	83
Involved Nodes			
Sending	100%	100%	72%
Receiving	100%	100%	72%
Overhearing	56%	56%	96%
Data Packet Latency [ms]			
Minimum	17	24	29
Lower Quartile	23	27	32
Median	26	31	37
Upper Quartile	35	38	45
Maximum	123	144	527

TABLE I
PACKET STATICSTICS

consumption is higher the closer a node is to the actual data path. This is caused by packet overhearing and internal protocol message exchange. In opposition to packet flow observed for Greedy Routing, nodes far apart from the data path still consume a small amount of energy for overhearing routing packets.

Another interesting aspect of routing protocols and sensor network applications is packet type distribution. Here, it is investigated which nodes in the network are sending or receiving certain packet types. The lower row in Fig. 11 displays packet types handled by all nodes in our example topology. In case of DSR, e.g., only nodes on the data path send and receive packet types 10 (route reply) and 12 (data packet). In contrast, all nodes either send or receive packet type 9, which corresponds to route requests. Note that packet reception here implies that a node receives a packet destined to itself or to the broadcast address.

Besides visual analysis, tables with descriptive statistics can also be derived from the generated message traces. Example data is shown in Table I. The number of packets sent reveals major differences between the protocols. Another comparison depicted in the table is the latency between data packet creation and their reception at the data sink. Here, it shows that TYMO produces the lowest latency. The high maximum latency of Greedy Routing is due to the fact that the implementation we used from [2] is a reactive one. In the absence of traffic, no neighborhood information is maintained. Whenever a node receives a packet, it produces a neighbor request. The node then gathers information about neighbor node positions in order to make the correct next routing decision. As pointed out in [2], this technique is a delay versus reliability tradeoff.

In conclusion, packet tracing enables protocol designers to gain an in-depth look at key metrics of their protocols. It additionally allows for easy comparison between protocols, whether using visual representation or plain figures – whichever seems more convenient or appropriate.

D. Overhead

The results in Table II are showing the overhead introduced of TinyAID. The simulations are based on the SelfWISE framework simulating 100 seconds, which means 100 rounds with one broadcast per round and node. The simulations are run on topologies from 4 up to 100 nodes. The nodes are arranged in a grid in such a way that a node in the center has exactly four neighbors. The simulations are performed with the original SelfWISE framework and three different instrumented versions. In the selective version exactly two functions are instrumented, namely those that are used to evaluate the results shown in Fig. 9. The partially instrumented code monitors all functions that are not part of the TinyOS operating system. In the complete instrumented version every function is instrumented. The overhead of simulation time is relative to the simulation time without instrumentation. The number of events represents the number of entering and leaving events. The results reveal that instrumentation must performed carefully. Otherwise the simulation time will increased up to three times. If only selective functions are instrumented, the introduced runtime overhead is not significant. The size of the resulting TOSSIM library depends on the number of instrumented functions. The selective instrumentation does not increase the size. The partially instrumentation increases the size from 552 kB by around 1% to 559 kB

Complete instrumentation leads to a non-functional executable, since the debug messages are printed before the simulation is setup. So at least `/opt/tinyos-2.x/tos/lib/tossim` must be excluded in order to simulate the instrumented code.

In our simulations of the routing protocols, sending 10 data packets from one corner of a 25-node network to the opposite one produced trace files with sizes between 10 and 20 kB. This amount could fairly be reduced by a more compact way of trace file layout. However, estimating log file size is difficult, because the number of packets sent in a network is the major driver here. It depends on the number of nodes; protocols used for medium access, routing, etc.; and the application.

Packet tracing causes a two-fold overhead. Firstly, additional code must be executed in order to log data. In the case of simulation, this overhead only concerns simulation execution time. In a real testbed, however, the processor of a node must execute the tracing code, which may, e.g., change timings or energy consumption. Secondly, packet size is increased by currently 4 Bytes. Depending on the data payload size, this increase may become significant and cause side-effects, such as increased packet transmission times or packet loss rates.

VI. RELATED WORK

The EvAnT framework [7] and the Rupeas language [8] are tools for sensor network analysis. Both approaches interpret a log file as an event collection with each row being an individual event. Each event is then specified by the column values. EvAnT and Rupeas feature event set processing, event set queries, and assertions for testing. The creation of log files, however, is not part of EvAnT and Rupeas. The system under consideration has to be instrumented manually with log file generating code first. In contrast, our approach supports automated instrumentation for creating log files. By providing

Number of Nodes	Without	Selective 2 Functions			Partially 101 Functions			Complete 183 Functions		
	Time	Time	Overhead	Events	Time	Overhead	Events	Time	Overhead	Events
4	0.45	0.46	0,03	1344	0.65	0,46	105866	1.58	2,54	615372
9	1.01	1.01	0,01	3024	1.58	0,57	303928	3.74	2,72	1506068
16	1.79	1.82	0,02	5376	2.94	0,64	605924	6.85	2,82	2793508
25	2.79	2.83	0,02	8400	4.71	0,69	998476	11.04	2,96	4465828
36	4.04	4.13	0,02	12096	6.89	0,70	1502018	16.15	3,00	6512520
49	5.52	5.62	0,02	16464	9.53	0,73	2102180	22.31	3,04	8992210
64	7.26	7.33	0,01	21504	12.70	0,75	2819526	29.33	3,04	11864168
81	9.18	9.28	0,01	27216	16.07	0,75	3608232	37.07	3,04	15132452
100	11.36	11.60	0,02	33600	19.93	0,76	4508578	46.45	3,09	18802426

TABLE II
EVALUATION OF THE OVERHEAD INTRODUCED BY TINYAID

the instrumenter with the right code templates, TinyAID is able to create any specific log file format, in particular those which might then be used for EvAnT and Rupeas.

The Sympathy [9] and Memento [10] network monitoring systems, and the concept of passive inspection [11] are focused on the message communication part of the system. In Memento and Sympathy the system under consideration is extended by additional code, which performs failure detection based on message monitoring. The same sensor network is then used to report logging data to a specific collecting node. Passive inspection of sensor networks follows a complimentary approach to Sympathy and Memento. Message log files are created by an additional deployment support network where every node owns two wireless transceivers. One transceiver is used in order to overhear all wireless sensor network traffic in the surroundings. The second transceiver, being a robust and high-bandwidth one, is used to transmit the results to a specific collecting node.

The presented TinyAID differs from Sympathy, Memento, and passive inspection. In these approaches system inspection always has to follow a black box approach, i.e., information about traffic patterns is used to infer information about code correctness on nodes. Moreover, there are no guarantees that all message communication failures are detected. In our approach code is instrumented directly at the points of message transmission. In addition, when a irregular behavior is observed, call-chain logging potentially supports finding the faulty module handlers directly.

Instrumentation approaches for wireless sensor networks are going beyond inspection of messages: EnviroLog [12] and Declarative Tracepoints [13]. In EnviroLog code has to be annotated manually first and then passed through a preprocessor before the final compilation pass. This preprocessing approach is comparable with the one presented by the TinyAID instrumenter. However, in contrast to EnviroLog, TinyAID exempts the programmer in many cases from touching the inspected code directly.

In the Declarative Tracepoint approach the high level declarative programming language TraceSQL for code instrumentation is introduced. Using this instrumentation language, the user is not required to manually touch any line of code under consideration. The language allows entering so-called action-associated check points into the source code. If the check point is passed and the check point predicate is satisfied, a certain action is performed. From the perspective of the language features, this approach appears to us the most general wireless sensor network instrumentation support. However, in contrast to the solution presented with TinyAID, the programmer has no control over what exactly is inserted into the instrumented code. It depends on the implementation of TraceSQL. In TinyAID code templates make the inserted code explicit and give the programmer full flexibility to tailor the templates to his needs. For example, the way logging is performed in a TOSSIM simulation may simply be done by using the `dbg` function, while logging in a real testbed deployment may be achieved by another code template for writing logging information to the serial port directly.

Support for message-flow tracing is another feature that distinguishes TinyAID from TraceSQL. While in TraceSQL check points may be added to the send/receive handlers of the Active Messaging module, there is no way to tag messages with an additional unique message identifier. In our approach unique message identification supports tracing message instances from the creating node to the nodes where the message is either dropped or delivered.

Other useful debugging approaches presented in the literature are NodeMD [14] and SNMS [15]. The emphasis of both approaches is a fine-grained node-level inspection of failure behavior. NodeMD focuses on specific node failures covering stack overflow, and lifelock/deadlock situations in multi-threaded environments. The goal is to catch such failures and provide the user with diagnostic information, which can then be used for troubleshooting, before the node becomes completely unusable. In SNMS the focus is on attribute export based on instrumenting the code manually. The variables that need to be exported must be tagged manually. Based on such tagged variables, the tool provides a query-based health data collection and persistent event logging system. Fine-grained node-level debugging versus inspecting system-wide behavior is the main difference between these approaches and the TinyAID approach presented in this work.

In this work we devised a major part on automated evaluation with visualization features that enables the developer to capture complex network-wide effects in a visual, intuitive

manner. To the best of our knowledge little effort has been spent in that direction so far. As an exception, the routing path visualization concept presented in the Rupeas publication [8] is close to the spirit of what we mean with capturing network-wide effects in an intuitive way. Compared to that visualization concept, we see the concepts presented in this work as a complement and an extension of a hopefully more and more growing set of available visualization concepts.

VII. Conclusion

In this paper a code instrumentation and evaluation tool for the TinyOS community is presented. We highlighted the advantages of automatic instrumentation support over manual instrumentation and presented a simple but effective way for automated code instrumentation. The so-instrumented code produces log information that covers two aspects: logs of the call chain and message flow. The automatic evaluation tool based on such logged data, enables a programmer to capture system-wide behavior in a visual and intuitive manner. In addition, statistical performance quantities can immediately be extracted from the generated log files in an uniform way.

The empirical studies performed with TinyAID show that automated instrumentation and evaluation are a valuable support for TinyOS-based programming. For instance, a synchronization problem in the SelfWISE simulation was figured out by just looking at the visual representation of an event trace. In addition, a set of routing protocols originating from different programmers are compared. Spending only little effort on reading the code, the protocols are evaluated, both visually and also by means of some plain statistical data.

The main empirical studies in this paper were performed by simulation only. The next development step is to add support for real sensor network deployments. It is to be considered how the logged data can be gathered in an effective way and how to order events system wide when node clocks are not synchronized. In our future work plan we envision a combined hard- and software approach that is currently in its initial design phase, based on the presented conceptual findings in this paper.

An additional, future project we see is the integration of other concepts into TinyAID, e.g., by supporting the appropriate log file structure TinyAID may be combined with Rupeas. Moreover, a future extension of TinyAID may be the use of the TraceSQL language concept. So far, TraceSQL did not leave us enough flexibility on which code is added as trace points; a reason why we chose the code template approach. In a future extension, however, TraceSQL and the concept of code templates might be combined.

Acknowledgment

The authors would like to thank René Steinrücken for his help during the implementation of the TinyAID instrumentation software.

References

[1] P. Buonadonna, J. Hill, , and D. Culler, "Active Message Communication for Tiny Networked Sensors," in *In Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (ICC '01)*, Anchorage, Alaska, USA, April 2001.

[2] H. Frey and K. Pind, "Dynamic Source Routing versus Greedy Routing in a Testbed Sensor Network Deployment," in *Proceedings of the 6th European Conference on Wireless Sensor Networks (EWSN '09)*, Cork, Ireland, Feb. 11–13 2009.

[3] C. Weyer and V. Turau, "SelfWISE: A Framework for Developing Self-Stabilizing Algorithms," in *Proceedings of the 16th ITG/GI - Fachtagung Kommunikation in Verteilten Systemen (KiVS '09)*, Kassel, Germany, Mar.2–6 2009.

[4] I. Chakeres and C. Perkins, "Dynamic MANET On-Demand (DYMO) Routing," http://tools.ietf.org/html/draft-ietf-manet-dymo-17, Mar. 2009, internet Draft, Version 17.

[5] D. B. Johnson and D. A. Maltz, "Dynamic Source Routing in Ad Hoc Wireless Networks," in *Mobile Computing*, Imielinski and Korth, Eds. Kluwer Academic Publishers, 1996, vol. 353.

[6] G. G. Finn, "Routing and Addressing Problems in Large Metropolitan-Scale Internetworks," Information Sciences Institute (ISI), Tech. Rep. ISI/RR-87-180, Mar. 1987.

[7] M. Woehrle, C. Plessl, R. Lim, J. Beutel, and L. Thiele, "EvAnT: Analysis and Checking of Event Traces for Wireless Sensor Networks." in *Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC '08)*. IEEE Computer Society, 2008, pp. 201–208.

[8] M. Woehrle, C. Plessl, and L. Thiele, "Poster Abstract: Rupeas - An Event Analysis Language for Wireless Sensor Network Traces," in *Poster/Demo Proceedings of the 9th European Conference on Wireless Sensor Networks (EWSN '09)*. Cork, Ireland: Springer, Feb. 2009.

[9] Nithya, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin, "Sympathy for the Sensor Network Debugger," in *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems (SenSys '05)*. New York, NY, USA: ACM, 2005, pp. 255–267.

[10] S. Rost and H. Balakrishnan, "Memento: A Health Monitoring System for Wireless Sensor Networks," in *Proceedings of the 3rd Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON '06)*, Reston, VA, September 2006.

[11] M. Ringwald, K. Römer, and A. Vialetti, "Passive Inspection of Sensor Networks," in *Proceedings of the 3rd IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS '07)*, 2007.

[12] L. Luo, T. He, G. Zhou, L. Gu, T. F. Abdelzaher, and J. A. Stankovic, "Achieving Repeatability of Asynchronous Events in Wireless Sensor Networks with EnviroLog," in *In Proceedings of the 25th IEEE Conference on Computer Communications (InfoCom '06)*, 2006.

[13] Q. Cao, T. Abdelzaher, J. Stankovic, K. Whitehouse, and L. Luo, "Declarative Tracepoints: A Programmable and Ppplication Independent Debugging System for Wireless Sensor Networks," in *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems (SenSys '08)*. New York, NY, USA: ACM, 2008, pp. 85–98.

[14] V. Krunic, E. Trumpler, and R. Han, "NodeMD: Diagnosing Node-Level Faults in Remote Wireless Sensor Systems," in *Proceedings of the 5th International Conference on Mobile Systems, Applications, and Services (MobiSys '07)*. ACM, 2007, pp. 43–56.

[15] G. Tolle and D. Culler, "Design of an Application-Cooperative Management System for Wireless Sensor Networks," in *Proceeedings of the Second European Workshop on Wireless Sensor Networks (EWSN '05)*, Jan.-2 Feb. 2005, pp. 121–132.