

Rupeas: Ruby Powered Event Analysis DSL

Matthias Woehrle, Christian Plessl, Lothar Thiele

Computer Engineering and Networks Lab, ETH Zurich

8092 Zurich, Switzerland

woehrle@tik.ee.ethz.ch

TIK Report 290

Updated Version 10.12.2009: Removed a comment footnote and corrected Typo in Figure 6.

February 18, 2010

Abstract

Wireless Sensor Networks (WSNs) are unique embedded computation systems for distributed sensing of a dispersed phenomenon. While being a strongly concurrent distributed system, its embedded aspects with severe resource limitations and the wireless communication requires a fusion of technologies and methodologies from very different fields. As WSNs are deployed in remote locations for long-term unattended operation, assurance of correct functioning of the system is of prime concern. Thus, the design and development of WSNs requires specialized tools to allow for testing and debugging the system. To this end, we present a framework for analyzing and checking WSNs based on collected events during system operation. It allows for abstracting from the event trace by means of behavioral queries and uses assertions for checking the accordance of an execution to its specification. The framework is independent from WSN test platforms, applications and logging semantics and thus generally applicable for analyzing event logs of WSN test executions.

1 Introduction

Testing of WSNs is a crucial part of the development process as a functional, performing and resource effective system at deployment time is of utmost importance. A main methodology for testing a WSN system is using a WSN specific test platform such as a testbed [4, 13] or simulator [8, 5] and logging test specific information. This logging information is used to validate the execution of the system. System tests result in a substantial amount of log data, necessitating adequate tools to analyze the logs considering the details of the test platform and the application. Analysis may be performed with mere scripting, which lacks formality.

A rigorous approach is needed for testing WSN systems facilitating the evaluation of logs and increasing reusability of analyses. Previously proposed state-based approaches [1] mainly targeted at single node software is such a semi-formal approach. However, for distributed software systems, which are loosely coupled but nevertheless highly integrated and operating cooperatively, a set of state machines is not sufficient.

In this work, we present a Domain Specific Language (DSL) for analyzing logs of WSN systems: The *Rupeas* language, a **Ruby Powered Event Analysis** facilitates analysis of WSN logs by using an *event abstraction* for each entry in the execution log. The

comprehensive log or *trace* corresponds to a (partially ordered) set of events. Rupeas provides operators to formulate queries on sets of events to extract behavioral information from an event set: For example, starting from individual send and receive events logged while routing packets, a Rupeas query can extract the routing paths of individual packets and determine whether and where a packet was lost along the way. Rupeas allows for formulating assertions on extracted information. Considering the previous example, a test run validating the network protocol may assert a failure condition if on an individual node more than 10 packets are lost.

Rupeas provides a high ease of use with its simple, concise notation by using a domain specific abstraction. It is targeted for integration into a larger testing [16] or analysis [2] framework by being embedded in a powerful scripting language. Being based on log files, the analyses are agnostic of actual test platforms and logging mechanisms. As such Rupeas allows for logging across differing test platforms, heterogeneous sensor networks, e. g. Trio [3], or sensor node operating systems. It greatly simplifies WSN trace analysis and testing, which are often perceived as meticulous tasks.

To this end, Rupeas provides the following novel contributions for testing and analyzing WSN systems:

1. Rupeas uses an event abstraction based on WSN traces, which allows for test and sensor node platform independent formulation of tests.
2. Rupeas allows for automated analysis and checking of event traces, which is indispensable for testing e. g. for regression or performance comparison.
3. Rupeas is a language especially designed for WSN event trace analysis. It facilitates writing analyses using a concise, declarative notation. Its embedding in a powerful host syntax allows for elaborate extensions and the integration into a comprehensive testing framework.

1.1 Related Work and Background

As an example, an interesting analysis of typical WSN applications for data gathering is whether and to what extent sensory data can be extracted from the system. Typical questions are: What is the data yield from each node? What do the routing paths look like? Are the routing paths efficient?

These are sample questions that can be answered by Rupeas queries. Rupeas allows for determining routing paths by formulating a relation on send and receive events gathered in traces, even for multi-sink systems as shown in Fig. 6 on page 13.

Previously proposed approaches cannot help with this analysis: Diagnostic simulation [7] using data mining techniques on simulation data, can help you with outlier detection, but does not provide a comprehensive analysis. Note also that outlier detection relies on statistics and learned good behavior for detection, while Rupeas looks at individual executions and determines success based on domain knowledge. As such Rupeas is similar to assertions on distributed, global state [6, 9]. Both of the global state approaches record traces during runtime and analyze them for properties. State-based approaches focus on snapshots of the distributed state, i. e. at a certain instant in time, while event based approaches analyze causal or temporal sequences, i. e. patterns, signifying a specific behavior. Another difference is that with assertions, the collection of information and analysis and oracles are tightly coupled. Even tighter integration is intended with Wringer [11], a debugging system running on individual nodes, utilizing

dynamic instrumentation and collecting global information through in-band communication. At its core a Scheme interpreter is used for evaluating debugging scripts, using predicates to determine localized state conditions.

Rupees takes a different approach: The monitoring and information collection is decoupled allowing for running different analyses on the same monitoring data and for integration into a comprehensive framework.

Complementary to analysis are test platforms for Wireless Sensor Networks, which can be chiefly categorized as simulators and testbeds. Simulators differ in their abstraction of motes, communication and environment. On the lowest level emulators such as Avrora [12] can be used. Code simulators such as the discrete event simulator TOSSIM [5] are based on simulating underlying hardware. High level simulators such as Castalia [8] abstract away from the actual nodes, but provide realistic radio models based on empirical models for specific radios. Most tools focus on execution on real hardware either on testbeds [6] or even at the deployment site [9].

2 Testing Wireless Sensor Networks

Validation of systems is a vital and substantial part in development. For autonomous WSN systems, which require to run without maintenance operations, deploying a functionally correct and performing system is of utmost importance.

Validation of sensor networks is difficult: The system is distributed, only loosely coupled thus exhibiting a large degree of concurrency. Verification of such distributed systems typically requires abstracted models to deal with the resulting complexity and the vast state space of the system. However, many of the issues concerning WSN system stem from the fact that the individual components of the system are embedded systems with stringent resource constraints for computation, memory and energy and necessitate tedious and error-prone event-driven, low-level programming. This is a fundamental crux: an abstraction to deal with the distributed complexity hides the complexity of embedded systems. Another fundamental problem is the bandwidth limitation of embedded systems: The lack of access to the device under test results in black boxes providing (almost) no insight in the internal operation. Instrumentation of the sensor nodes try to accommodate for this problem, but suffer from probing effects.

Testing is the de facto standard for software validation. While testing can merely show the presence of errors and not their absence due to sampling of distinct test scenarios, it allows for detailed and accurate models of the system, its constituent sensor nodes and the environment the system is integrated into.

2.1 Testing and test automation

Testing is a continuous task: Tests provide a safety harness, which validates changes as well as novel features. Hence, tests need to be automated to allow for repeated execution. Frameworks for continuous integration and unit testing provide the skeleton of the test infrastructure in that they allow to setup and execute tests and present results in a concise, user-readable format. For WSN testing, automation of tests requires a dedicated WSN testbed integrated into a comprehensive test architecture [14]. However, testing of a comprehensive system test requires more than individual, separated assertions on test success on individual nodes. Test oracles determining test success need to analyze the global execution of a system. Large systems and long monitoring intervals result in a plethora of information provided by the testbed. This information

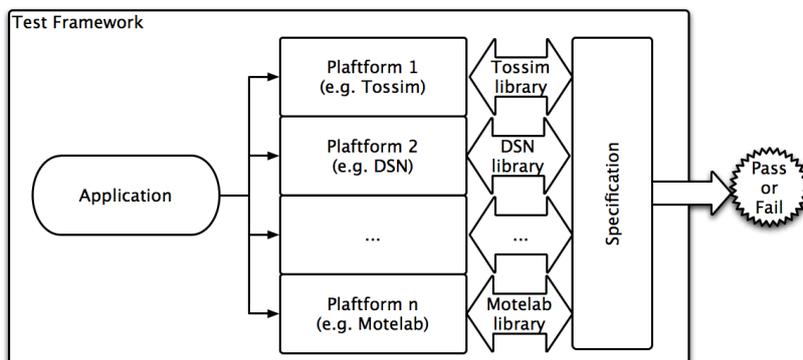


Figure 1: Multi-platform testing using platform APIs [16] for platforms such as Tossim [5], DSN [4] or Motelab [13].

necessitates an automated analysis for asserting a valid test execution based on a test oracle.

2.2 Test platforms

The challenges of WSN system design has resulted in an increased interest in different test platforms. For simulation this ranges from custom distributed simulations of algorithms, to network simulators with radio models in differing detail, to discrete event simulators, to WSN simulators down to software emulators. On the hardware side, test platforms start at individual nodes placed on a desktop, to testbeds featuring hundreds of nodes. Differing logging mechanisms and protocols exist for each of the different test platforms. Additionally, systems typically feature different node platforms: A tiered architecture with a more powerful gateway has been a standard implementation since the first deployments, e. g. on Great Duck Island [10].

When developing a system, testing is performed on differing test platforms. Starting from high level simulation, the development gradually transfers to integration tests on a full-fledged testbed. Tests may also be performed on different abstraction levels trading off accuracy for test execution speed. All tests on the different platforms require analysis of the test execution in form of logged instrumentation traces. When performing tests on differing platforms, the question of reusability of tests across platforms is a major concern. In our previous work we have proposed a general test API for differing test platforms to allow for multi-platform testing frameworks [16] as shown in Fig. 1. A fundamental issue for WSN testing is the logging and analysis of distributed execution and an extraction into global behavioral information. A basic denominator for logging is that all test platforms provide some notion of a type of an event, which happened on a distinct sensor node. The extraction of behavioral information requires novel techniques as the interaction of distributed system components differs to traditional distributed systems providing reliable communication and a larger visibility into the system.

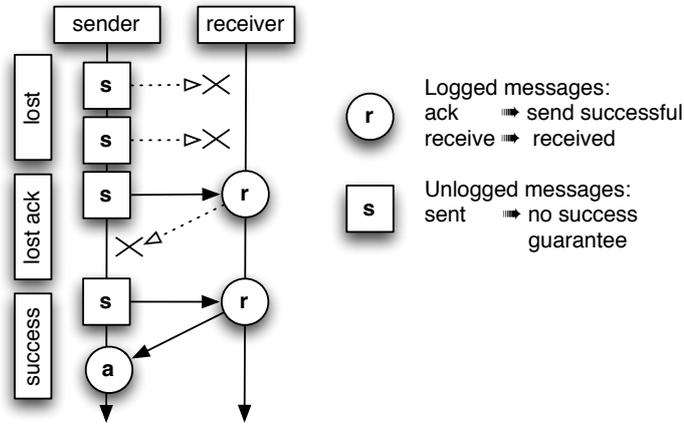


Figure 2: Message semantics

2.3 Wireless communication implications

WSN systems communicate over the ether. There are several reasons why a packet, which is transmitted might not be received: Noise, interference, collisions, channel effects and radio duty cycling (i. e. the receiver is sleeping and does not listen). Lost packets can only be detected when packets are acknowledged. A missing acknowledgement is a necessary condition for a lost packet and used as an indicator. It is not sufficient as the acknowledgement itself might be lost, while the sent packet was received correctly. Handling of lost packets is application and instrumentation-specific. Thus there is no direct relation between send and receive events. An analysis of events collected during system execution must provide a flexible framework for specifying different kinds of message semantics as shown in Fig. 2.

3 Traces, Events and Event Analysis

Events refer to instantaneous information provided by the system while executing. We define an event e as a single instance produced by an instrumentation on one of the k sensor nodes in the system, which we denote as n_1, \dots, n_k .

Events can be:

- state change induced by local code execution such as a variable assignment or function call
- state change induced by an external event such as a message either sent or received
- a (periodic) snapshot of current state

Events are collected from individual nodes and merged into a comprehensive trace as shown in Fig. 3.

Visibility of this information may be established in different ways: hardware or software monitors or passive monitoring. Passive monitoring [9] requires additional

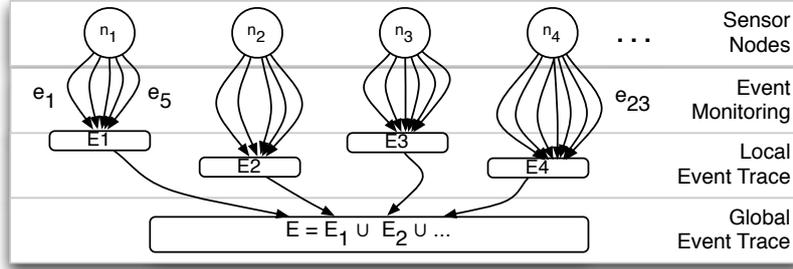


Figure 3: Trace

compensation due to its intrinsic unreliability. Unobtrusive hardware monitors are expensive and require physical access. Event generation and efficient logging is out of scope of Rupeas, as it is focussed on the analysis of given execution traces and agnostic of the generation of the trace.

3.1 Event definition

Definition 3.1 An event e is a n -tuple of key-value pairs, which minimally comprises a node identifier signifying the origin of the event and a type identifier to classify the event.

Further attributes of an event are added by key-value-pairs.

$$e = (\text{node} : \text{node}_{id}, \text{type} : \text{type}_{id}, \text{key}_1 : \text{value}_1, \dots)$$

Events of a single type have a consistent format.

Example 3.1 A reboot event logged on node n_{10} at time $t = 10789$ is represented by the 3-tuple:

$$(\text{node} : n_{10}, \text{type} : \text{reboot}, \text{time} : 10789)$$

Note that $n \geq 2$, since a node identifier and a type identifier are minimally required for any event. Events may be test-, application- or test platform specific.

3.2 Traces are event sets

Events are logged for any instrumented node n_i . The collection of events is an event set E_i , which is the trace of all events occurred during the execution of the system. The global trace is the event set E , which is the union of the distributedly collected event sets.¹ A trace is a partially ordered. Local events (on an individual sensor node) are totally ordered by the sequential execution of tasks on each sensor node. However, events on different nodes may be concurrent. Order information is inherent in the event attributes such as recorded timestamps or the logical which can be constructed from the local order and inter-node communication.

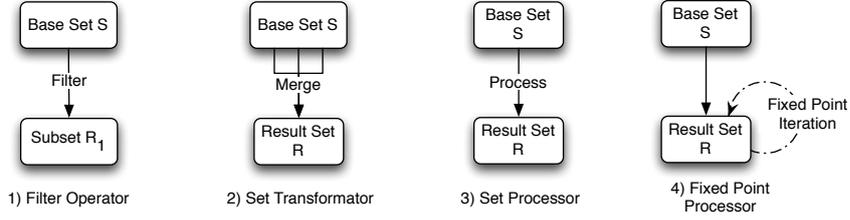


Figure 4: The four operators for an event analysis

3.3 Operators

Event analysis uses events and event sets as primitives for the analysis of a system execution. Thus, event analysis offers the set operations of union, intersection and relative complement. Additionally, event analysis offers four novel operators (cf. Fig. 4) especially tailored for formulating queries on WSN event sets. As Fig. 4 indicates, S is the set used as the input for the operators. R is the result set, i. e. the output, of a given operator. s denotes an event in the base set S . The event analysis operators are based on a *selection predicate* φ and a *transformation function* f .

Definition 3.2 (Selection Predicate) A selection predicate $\varphi(e_1, e_2, \dots), e_i \in S$ is defined on one or multiple events and uses relations on the values of specified event keys for selecting events from a base set S .

Predicate relations differ based on the purpose of the selection and the available trace information.

Example 3.2 (Selection predicate) In order to select a reboot event from the event trace, the following predicate is used:

$$\varphi(s) := s.type == reboot$$

Predicates may also be defined on multiple events, e. g. to define a send-receive relation:

$$\varphi(A \in S) := \varphi(r, s) := r.type == receive \wedge s.destination == r.nodeid \wedge s.type == send$$

Definition 3.3 (Transformation function) A transformation function is a function $f(A) : 2^S \rightarrow 2^S$ on an event set A returning an event set A' .

It is used to either (1) add information to events in A or to (2) merge the events in A into a compound event object.

Example 3.3 (Transformation function) (1) For a hop-count analysis, we need to add a hop-count field to individual events. Hence we define the following transformation function on an event e :

$$f(e) = e | e.hopcount = 0$$

(2) In order to join a send and its according receive event into a compound event (e. g. e_{11} and e_{21} in Fig. 5), a transmission, the event is transformed to maintain information from both events $a1 = (node = n_i, type = send), a2 = (node = n_j, type = receive)$:

$$g(a1, a2) = e | e.node = a2.node \wedge e.type = transmission \wedge e.sender = a1.node \wedge e.receiver = a2.node$$

¹We denote sets with upper-case letters, while events are denoted with lower-case letters.

The four novel event analysis operators are defined as follows:

Definition 3.4 (Filter Operator) *The filter operator allows to select a set of events into a subset based on the values of event keys. Filtering is performed on a single set S . The operator returns a single sets R , containing the events that satisfy a given predicate $\varphi(s)$.*

$$R = \{s | s \in S \wedge \varphi(s)\} \quad (1)$$

Definition 3.5 (Set Transformator) *The set transformator allows to select a subset A from a base set by using a predicate $\varphi(A)$. Selected events are processed based on a transformation function. Processed events are added to the result set R .*

$$R = \{e | e \in f(A) \wedge A \subseteq S \wedge \varphi(A)\} \quad (2)$$

Processing is performed on a single set. An extension to use the operator on multiple sets is to use the union of the sets as the base set and describe a predicate that discriminates individual events based on their origin set.

The set transformator operates on a selected subset of events. On the other hand, the set processor operations on each individual event in a set S .

Definition 3.6 (Set Processor) *Each event $s \in S$ is selected and processed by a transformation function $f(s)$ on the event. This allows for adding key-value pairs for events or computations on event values.*

$$R = \{e | e \in f(s) \wedge s \in S\} \quad (3)$$

Definition 3.7 (Fixed Point Processor) *The Fixed Point Processor computes the least fixed point of a given function on event sets and produces a new set. Selection and processing of events in a single iteration is performed as in the Set Transformator. Iteratively the sets R^i are computed, until a fix point is determined, i. e. $R^k = R^{k-1}$. All events that are not selected by the predicate are maintained for each iteration.*

$$\begin{aligned} R^0 &= S \\ R^i &= \{ \{e | e \in f(A) \wedge A \subseteq R^{i-1} \wedge \varphi(A)\} \cup \\ &\quad \{e | e \in A \wedge A \subseteq R^{i-1} \wedge \neg \varphi(A)\} \} \end{aligned} \quad (4)$$

A main application of the Fixed Point Processor is to determine routing paths as shown in Fig. 5. Iteratively, send and receive events are joined satisfying a given predicate such as having the same origin and the same sequence number.

Taking a closer look at the fixed point operator reveals that it iterates over all event $s \in S$ in order to find selection matches. However given a predicate ϕ , the operation can be optimized by pre-filtering the set S for a given start event s_0 . Considering the example above, the fixed point operator operating on the routing paths from $n0$ can create a subset $S_0 = s | s \in S \wedge s(\text{origin}) = n0$ considerably speeding up the matching process.

4 Rupeas

Rupeas (**R**uby **p**owered **e**vent **a**nalysis) is a language specifically designed for the analyzing event logs of WSN executions: it is a Domain Specific Language with a dedicated purpose. As such Rupeas leverages the event abstraction for WSN execution

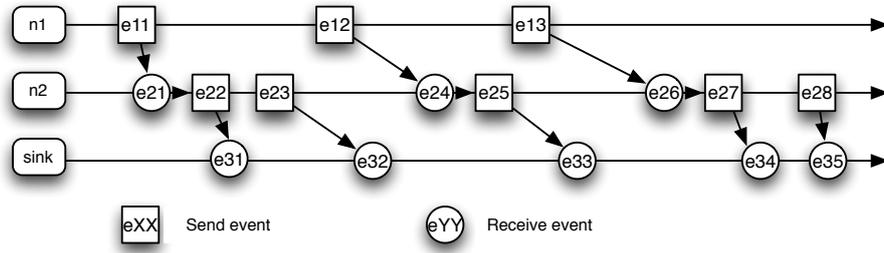


Figure 5: Using send and receive events to determine the routing paths

```
require 'Rupeas.rb' #Load rupeas language definition
Rupeas.new do
  #Sandboxed Rupeas Context
end
```

Listing 1: Rupeas Context

logs to process event sets and extract and verify behavioral information intrinsic to the traces.

Rupeas is targeted for test automation for differing test and mote platforms. A major design goal is the integration into a larger testing framework [16]. To this end, Rupeas is implemented inside a powerful dynamic language: Ruby. Ruby's flexible syntax allows for easy formulation of so-called internal DSL's. As such a user has a three-fold benefit of using Rupeas: 1) The event analysis is written in a specifically designed concise language with clear semantics and reduced syntactical noise. 2) Supplementary processing or analyses can be formulated in Ruby. 3) Rupeas is valid Ruby syntax and thus immediately usable on platforms providing Ruby interpreters such as Linux and MAC OS X merely by inclusion of the Rupeas library.

The Rupeas DSL uses a separate context to define the starting point of an event analysis. Listing 1 shows the inclusion of the Rupeas library and the creation of the context. The context allows for a sandboxed execution of Rupeas scripts.

4.1 Event Declaration

In Rupeas, users declare each event in a trace to be analyzed. Using the event declaration, Rupeas automatically parses the event traces in a flat file format with individual event entries as displayed in Listing 2. Each event lists its properties, with the first column signifying the event type.

Rupeas declared event types in a *Type* block: Each property of a type is specified by its name, its type, acceptable values and notification levels on outliers. Types include the basic types with the addition of a periodic type, which can be used for wrapping integers commonly found in embedded systems e. g. for timers. Notification levels are warnings and errors, where warnings only display a warning message, while errors stop the analysis. As an example, a *senddone* event as shown in Listing 3 specifies that each *senddone* entry is followed by a periodic-type sequence number in the range of 0 to 255. In case the sequence number is outside this range, Rupeas stops the analysis with

```

...
senddone 53 21 0 41 4111.546244
receive 53 60 0 0 4111.564199
senddone 53 60 0 52 4111.564367
receive 53 11 41 41 4111.570129
senddone 53 11 41 31 4111.570297
...

```

Listing 2: Event trace excerpt

```

Type :senddone do
  with :name =>:seqNo, :fieldtype=> :periodic, :range => 0..255, :
  -notification => :error
  with :name =>:origin, :fieldtype => :integer, :range => 0..100,:
  -notification => :error
  with :name =>:destination, :fieldtype => :integer, :range => 0..100, :
  -notification => :warning
  with :name =>:nodeid,:fieldtype=> :integer, :range => 0..100,:
  -notification => :error
  with :name =>:time, :fieldtype => :float, :notification => :warning
end

```

Listing 3: Declaration of Send events

an error assertion.

4.2 Event set Queries

Filtering in Rupeas is as simple as selecting the events from the set. Either, events are selected based on having a specific key or by the value of a specific key as shown in Listing 4.

Queries in Rupeas, i. e. set processor, set transformers and fixed point processors, are block structured. A block transforms the event set it is used on. Set processor and set transformer are only distinguished by the number of events passed to the block. In order to use the fixed point processor an `:iterative` parameter is passed to the transformation. When transforming an event set, selection predicates and transformation functions are encapsulated in an associated block. The block specifies the number of events to be processed and provides the naming for the block allowing for direct association of the selection predicate and the transformation function.

Predicates are specified as constraints (*constraint*) or selections (*select*). Constraints

```

originevents = all[:origin]           #select all events having an :
→origin key
sends = originevents[:type=>:senddone] #select all events having an :type key
→with value :senddone

```

Listing 4: Filtering out all `:senddone` events from an eventset all

```

#Setup
routestart = all.transform do |sending|
  constraint sending[:origin] == sending[:nodeid] # splitting into subsets
  constraint sending[:type] == :senddone # filter
  merging :type => :route, :nodeid => sending[:nodeid], :seqNo => sending[:seqNo], :origin => sending[:origin]
end

# Iterate
routes = routestart.transform(:iterative) do |sending, receive|
  constraint sending[:origin] == receive[:origin]
  constraint sending[:seqNo] == receive[:seqNo]
  select sending[:nodeid] == receive[:nodeid] and receive[:type] == :route and sending[:type] == :senddone
  select sending[:dest] == receive[:nodeid] and sending[:type] == :route and sending[:type] == :receive
  merging :type => :route, :nodeid => receive[:nodeid], :seqNo => sending[:seqNo], :origin => sending[:origin]
end

```

Listing 5: Set up route anchors on origin nodes and iterating over paths

are predicates, which hold globally and thus can be used internally for splitting the set for computational speed-up. Selections are predicates, which depend on the current event to be matched. Each selection is an individual option, i. e. allowing only for conjunction of terms. Selections are composed by disjunction into a compound predicate. As an example shown in Listing 5, when determining the actual route in the Iterate step, the association of individual send and receive events depends on the current events in the set: In Fig. 5, e_{11} , a `:senddone` event is first matched with e_{21} , a `:receive` event, forming a `:route`. This `:route` event is subsequently matched with the `:senddone` event e_{22} and so on. Notice in this case, the selection of a matching event is performed in each iteration. On the other hand, it must always hold that sequence number and origin of packets match. Hence, this is specified as a constraint in the Iterate step of Listing 5.

Transformation functions are specified in terms of resulting events: Each event generated from a match is specified in terms of key-value pairs of the constituent selected events. As an example, Listing 5 shows how the start of a packet route is determined. First, the event must indicate a sent packet (`:senddone`) and secondly, the origin of the event must match its node identifier. The transformation function maintains the inherent information in the event, but changes the type to `:route` to mark the route start.

4.3 Lessons Learned from EvAnT

EvAnT [15] was a first approach to encapsulate the concept of event analysis into a testing framework. Rupeas provides the following advantages and extensions to EvAnT.

1. Consistent data types
EvAnT provided a partition operator returning a set of events. Hence, there is an additional data type in the event analysis format, which renders the processing more difficult. Rupeas provides a filter operator returning a single event set. Thus

Rupeas only considers events and event sets, which provides clearer separation from the host language.

2. Declaration of events

Event collection and preparation is a considerable part of an analysis. Thus, EvAnT showed that the event definition can provide from a declaration of events and their properties and constraints to validate event traces. This additional feature is added to Rupeas with its declarative event type declarations as depicted in Listing 3. Event declarations allow for ensuring the validity of events in the event traces before starting the analysis. Typical testbed problems such as missing nodes, which failed to log during the test can be detected. One specific feature is a new data type provided for periodic value types for events, which are used for wrapping counters such as sequence numbers. Rupeas features a heuristic allowing for determining a total order on these values.

3. Lists, sets, posets

Event sets in Rupeas are ordered sets. However, general event sets for distributed computations are partially ordered sets. The implementation favors the use of totally ordered sets or lists, however providing distinct sorting and comparison operators supporting the partial order semantics.

4. Predicates, transformation functions and query complexity

Query complexity due to intricate predicates and long transformations in the form of anonymous functions complicate EvAnT. To this end, in Rupeas we take a different, more declarative approach: A single query is a block, which encapsulates the predicate and the transformation function. This facilitates writing and understanding and is reused internally for optimizations of global predicates. Transformation functions are restricted to declaration of new events based on the constituent selected events. Predicates and transformation functions are tightly coupled by sharing event selector parameters as shown with *sending* in Listing 5.

4.4 Rupeas language specification

Listing 6 presents the EBNF for the Rupeas language in the Appendix.

5 Case Study

As an example for an analysis of a data gathering application, we apply Rupeas to data from simulating a TinyOS 2.x application. We instrument the Collection Tree Protocol to log sent and received packets and simulate the system in TOSSIM [5]. The simulation topology is a 70 node grid (cf. Fig. 6) including two sink nodes (0, 100). The simulation uses gain and noise models based on USC's Realistic Wireless Link Quality Model and Generator². The log file and input for Rupeas captures data from a 6 hour run resulting in over 2 million events.

The analysis features two basic steps: Loading the event trace using the event description (cf. Listing 3) and using the fixed point processor (cf. Listing 5). Rupeas allows for filtering routing paths for final destination, route selection or hop count. The data can be easily extracted with Ruby into different file formats. The main results obtained

²<http://anrg.usc.edu/www/index.php/Downloads>

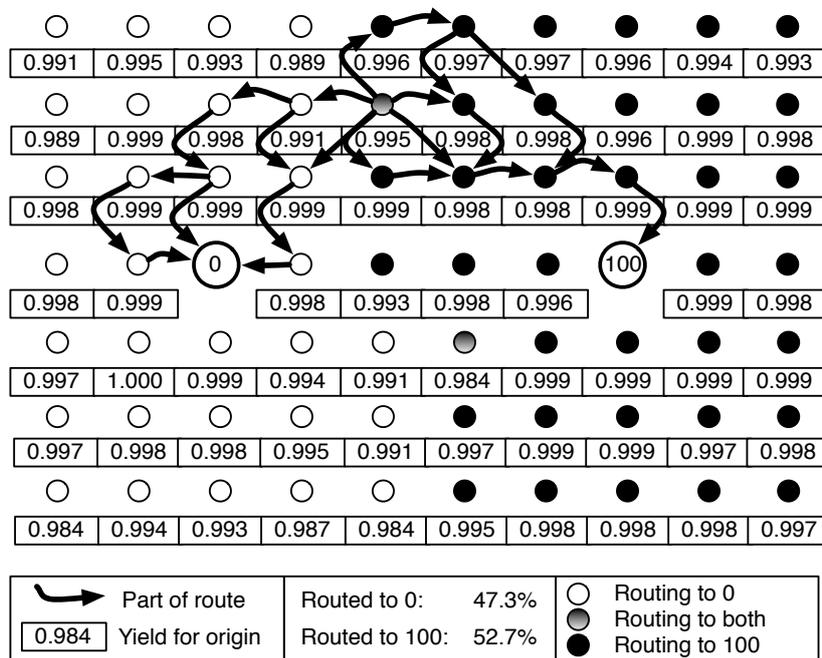


Figure 6: Visualization of Rupeas results.

are visualized in Figure 6: It depicts the yield of individual origin nodes and their associated sink. The data yield of the simulation is high and the traffic is routed evenly among the sinks. As an example, The figure also displays the links of nine different routing paths actually taken in the experiment for node 24: The packets are routed via minimally two and maximally six intermediate hops. 50.4% of the packets are routed to sink 0.

This analysis is only a single example of Rupeas capabilities.

6 Summary

Rupeas is an internal DSL in Ruby for analyzing WSN test log files. It can also be used to analyze the maintenance logs typically collected during actual deployments. It provides a clear syntax based on a domain abstraction of event sets. Its embedding in Ruby allow for embedding in test frameworks, analyses or maintenance routines. Rupeas is an open source project. Further details on Rupeas are available on the Rupeas project page³.

7 Acknowledgments

The work presented here was supported by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322.

³<http://code.google.com/p/rupeas/>

References

- [1] James H. Andrews and Yingjun Zhang. General test result checking with log file analysis. *IEEE Transactions on Software Engineering*, 29(7):634–648, 2003.
- [2] F. Zhao; J. Liu D. Chu and M. Goraczko. Que: A sensor network rapid prototyping tool with application experiences from a data center deployment. In *Proc. 5th European Workshop on Sensor Networks (EWSN 2008)*, 2008.
- [3] Prabal Dutta, Jonathan Hui, Jaein Jeong, Sukun Kim, Cory Sharp, Jay Taneja, Gilman Tolle, Kamin Whitehouse, and David Culler. Trio: enabling sustainable and scalable outdoor wireless sensor network deployments. In *Proc. 5th Int'l Conf. Information Processing Sensor Networks (IPSN '06)*, pages 407–415, 2006.
- [4] M. Dyer, J. Beutel, L. Thiele, T. Kalt, P. Oehen, K. Martin, and P. Blum. Deployment support network - a toolkit for the development of WSNs. In *Proc. 4th European Workshop on Sensor Networks (EWSN 2007)*, pages 195–211, 2007.
- [5] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proc. 1st ACM Conf. Embedded Networked Sensor Systems (SenSys 2003)*, pages 126–137, November 2003.
- [6] M. Lodder, G. Halkes, and K. Langendoen. A global-state perspective on sensor network debugging. In *Proc. 5th IEEE Workshop on Embedded Networked Sensors (HotEmNets 2008)*, 2008.
- [7] Mohammad Maifi, Hasan Khan, Tarek Abdelzaher, and Kamal Kant Gupta. Towards diagnostic simulation in sensor networks. In *Distributed Computing in Sensor Systems*, pages 252–265. Springer, 2008.
- [8] National ICT Australia. National ict australia - castalia [home], September 2007.
- [9] Kay Römer and Matthias Ringwald. Increasing the visibility of sensor networks with passive distributed assertions. In *Proc. 4th Workshop on Real-World Wireless Sensor Networks (REALWSN '08)*, 2008.
- [10] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler. Lessons from a sensor network expedition. In *Proc. 1st European Workshop on Sensor Networks (EWSN 2004)*, pages 307–322, 2004.
- [11] Arsalan Tavakoli, David Culler, Philip Levis, and Scott Shenker. The case for predicate-oriented debugging of sensornets. In *Proceedings of the 5th Workshop on Hot Topics in Embedded Networked Sensors*, 2008.
- [12] Ben L. Titzer, Daniel K. Lee, and Jens Palsberg. Avrora: scalable sensor network simulation with precise timing. In *Proc. 4th Int'l Conf. Information Processing Sensor Networks (IPSN '05)*, page 67, 2005.
- [13] G. Werner-Allen, P. Swieskowski, and M. Welsh. MoteLab: A wireless sensor network testbed. In *Proc. 4th Int'l Conf. Information Processing Sensor Networks (IPSN '05)*, pages 483–488, April 2005.
- [14] M. Woehrle, J. Beutel, and L. Thiele. Wireless sensor networks testing and validation. In *Handbook of Embedded Systems*, 2009.

- [15] M. Woehrle, C. Plessl, R. Lim, J. Beutel, and L. Thiele. EvAnT: Analysis and checking of event traces for wireless sensor networks. In *Proc. IEEE Int. Conf. on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC2008)*, pages 201–208, June 2008.
- [16] Matthias Woehrle, Christian Plessl, Jan Beutel, and Lothar Thiele. Increasing the reliability of wireless sensor networks with a distributed testing framework. In *Proc. 4th IEEE Workshop on Embedded Networked Sensors (EmNetS-IV)*, pages 93–97. ACM, 2007.

A EBNF definition for Rupeas DSL

```
* rupeas_program          ::= ruby_code | event_description |
-rupeas_processing

* event_description      ::= "Type" Symbol block
* block                  ::= do_block | curly_block
* do_block               ::= "do", event_field_descr, "end"
* curly_block           ::= "{", event_field_descr, "}"
* event_field_descr     ::= name , {constraint}, [notification_type]
* name                   ::= ":name" => Symbol
* constraint             ::= fieldtype_constraint range_constraint,
-notification_type
* fieldtype_constraint  ::= ":fieldtype" => ":integer" | ":float" | ":
-string" | ":periodic"
* range_constraint      ::= ":range" => Array| Range| bounds
* bounds                 ::= "{[_upper_=>]", upperboundvalue , "
-_lower_=>]", lowerboundvalue , "}"
* upperboundvalue       ::= Fixnum | Float
* lowerboundvalue       ::= Fixnum | Float
* notification_type     ::= ":notification" => ":error" | ":warning"

* rupeas_processing     ::= load_set | event_set_processing |
-event_assignment
* load_set               ::= event_set "=" "loadlog("filename"
-)"
* event_assignment      ::= event_set "=" event_set_processing
* event_set_processing  ::= event_set".transform"[(option)] process_block
* option                 ::= ":iterative"
* process_block         ::= "do_" process_variables, constraints,
-selectors, mergers, "end"
* constraints            ::= {"constraint" predicate}
* selectors             ::= {"select" and_term}
* and_term               ::= predicate {"and" predicate}
* predicate              ::= process_variable
-relation_operator process_variable | Value
* mergers                ::= "{merging" event_keys =>
-event_values}
```

Listing 6: Rupeas BNF