Rupeas: Ruby Powered Event Analysis DSL

Matthias Woehrle*,Christian Plessl†,Lothar Thiele*
*Computer Engineering and Networks Lab, ETH Zurich, Switzerland
†University of Paderborn, Paderborn Center for Parallel Computing, Germany

Abstract—**Wireless Sensor Networks (WSNs) are unique embedded computer systems for distributed sensing of a dispersed phenomenon. 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. To this end, we present a novel language, Rupeas, for analyzing WSNs based on collected events during system operation. Rupeas is independent from test specifics and thus generally applicable for analyzing event logs of WSN test executions.**

I. Introduction

Testing of WSNs is a crucial part of the development process as a functional, performing and resource efficient 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 [1] or simulator [2] and logging test specific information.

A vital part of a test is the analysis of a *trace*, the information logged from a system during its execution. However, this information is dependent on: (i) the specific application, (ii) what information is logged on the sensor nodes and (iii) the test platform used. The analysis is chiefly dependent on so-called *logging policies*, i. e. which data is logged and its meaning w. r. t. the application. As an example the relation between send and receive events may be studied. A transmission typically associates a sent log message on a node and a received log message on the intended receiver. However, depending on the logging policy, i. e. where the sending of the message is monitored, the resulting log differs considerably. Logging on the link layer results in sent messages, which have no corresponding received messages, since a transmission may fail. On the network layer a message is only logged as sent if the message has actually been received on the receiver node, i. e. there is a one-to-one correspondence. On the transport layer only the final destination node logs a receive indication. Hence, the analysis needs to consider the logging policy and allow for describing the semantic information of logged information. Since tools and languages need to adapt to the employed logging policy, in practice ad-hoc scripts are often used for analyzing execution logs. However, for increasing reusability of analyses across different tests and test platforms, a more rigorous approach is needed.

We previously presented a generic framework for the analysis of WSN logs allowing independent of logging policies, test and test platforms based on an event analysis [3]. In this work, we leverage this event analysis framework to create a Domain specific language (DSL) for analyzing logs of WSN systems: *Rupeas*, a **R**uby **P**owered **E**vent **A**nalysi**s**.

Rupeas design goal is to provide a simple, concise notation by using the domain specific abstraction of event analysis. Rupeas provides the following contributions:

1) It allows for automated analysis of execution logs.
2) It uses an event abstraction, which allows for test-, logging policy and platform-independent formulation of tests.
3) Rupeas is a language especially designed for WSN event trace analysis. It facilitates writing analyses using a concise, declarative notation.

Rupeas does not impose any structure on the monitoring and its formats. Moreover, it is targeted for integration into a larger test [4] or design [5] framework. Hence, Rupeas needs to be integrated with a scripting language allowing for typical testing tasks such as test automation, data processing or visualization. To this end, Rupeas is integrated with a powerful scripting language: Ruby.

II. Domain Specific Language

Event analysis is an abstraction using the notion of events and event sets [3]. This domain specific abstraction can be used for formulating a Domain Specific Language (DSL). The basic idea of a DSL is to facilitate the use of a language by focussing on the domain-specific tasks, here processing of event sets, rather than providing a general purpose language. Examples for DSLs include the query language SQL, `make` or `ant` for building software or CSS for web design. DSLs are categorized based on whether they have a custom syntax such as SQL or `make`, or if they use the syntax of a host language. Due to its embedding, Rupeas can leverage Ruby's flexible syntax, which facilitates developing internal DSLs with a large degree of freedom on program structure.

Nevertheless, Rupeas is a language specifically designed for the analyzing event sets: it is a DSL with a dedicated purpose. As such Rupeas leverages the semantic model of event analysis in order to process event sets and to extract behavioral information in order to analyze test executions. Using Rupeas provides several benefits: (i) The event analysis is written in a specifically designed concise language with clear semantics and reduced syntactical noise, (ii) supplementary processing or analyses can be formulated in Ruby, (iii) Rupeas can be easily extended in Ruby, e. g. for plotting results[1] or statistical analysis[2], (iv) Rupeas is valid Ruby syntax and thus immediately usable on platforms providing Ruby interpreters

[1]http://rgnuplot.sourceforge.net/
[2]http://rubyforge.org/projects/rsruby/

such as Linux and Mac OS X, and (v) with merely adding a simple event parser, Rupeas is usable for any project.

III. Language implementation

The Rupeas DSL uses a separate context to define the starting point of an event analysis. The listing below shows the inclusion of the Rupeas library and the creation of the context [6], providing a sand-boxed execution of Rupeas scripts, i. e. Rupeas functionality is only valid within the context for event declaration and processing.

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

A. Event Declaration

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

```
...
senddone 53 21 0 41 4111.546244
receive 53 60 0 0 4111.564199
senddone 53 60 0 52 4111.564367
...
```

Rupeas declares 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 (integer, float and strings) with the addition of a `:periodic` type, which can be used for wrapping integers, e. g. counters, commonly found in embedded systems. Notification levels for input trace format problems include the `:warning` and the `error`. Warnings only display a message and the cause for the warning, while errors stop the analysis. As an example, a `:sendone` event type definition as shown in Listing 1 specifies that each `:senddone` key is followed by a `:periodic`-type sequence number in the range of 0 to 255. In case the there is an event with a `:seqNo` outside the range $0 \ldots 255$, Rupeas stops the analysis with an error assertion.

```
Type :senddone do
 with :name =>:seqNo, :fieldtype=> :periodic,
 →:range => 0..255, :notification => :error
 ...
end
```

Listing 1. Type declaration of `:senddone` events

B. Processing event sets

Rupeas is based on event operators for processing event sets. It provides: (i) a filter operator to select a set of events into a subset based on the values of event keys, (ii) a set transformator selecting a subset A from a base set by using a predicate $\varphi(A)$ and processing A based on a transformation function. An iterative application of the set transformator computing the least fixed point of a given function on event set, called Fixed point processor. For definition and details on these operators the interested reader is referred to [3], [6].

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 below. From a set of `all` events only the (communication) events featuring an `:origin` key are selected. Subsequently the events are filtered, which feature a `:senddone` type.

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

Listing 2. Filtering `:senddone` events from event set `all`

All other operators are applied to a given set, e. g. the `all` event set in line 2 of Listing 3. In this case the set transformator (`:transform`) returns a set, which is assigned in this case to `routestart`. It is specified as a block, with the number and name of events it processes, here only `sending`. The selection predicate is specified using two constraints on the sending event. The start of a routing path is selected, hence the event must be a `:senddone` event and its node identifier must match the origin of the packet. If such a route start event is selected, its type gets changed to `:route`. As described before, a fixed point processor is similar to a set transformator but iteratively performs the transformation until a fixed point is reached. Hence in Rupeas the only difference of a fixed point processor to a set transformator is an `:iterative` indication on the according code block as shown in line 9 of Listing 3.

Selection predicates are differentiated depending on their usage: *Constraints* (`constraint`) are global invariants on the set. *Selections* (`select`) specify predicates, which depend on the the specifics of the events in the set. Selections only allow for conjunction of terms and may be composed by disjunction into a compound predicate. An example for selections is shown in Listing 3 for 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 Listing 3, a `:senddone` event is first merged with a `:receive` event, forming a `:route` (lines 2-6). This `:route` event is subsequently merged with another `:senddone` event (lines 9-15) and so on. Notice in this case, the selection depends on the current state of the processing of the set. As an example for a global invariant, the packets must always feature the same sequence number and origin. Hence, this is specified as a constraint in the `:iterative` step of Listing 3.

Transformation functions are restricted to only generate new events, e. g. based on the events selected by the predicate. These new events may either be simply added (in Rupeas by using the keyword `create`) or replace existing events (using `merge`). Transformation functions are specified in terms of resulting events: Each event generated is specified in terms of key-value pairs of the constituent selected events. As an example, Listing 3 shows how the start of a packet route

```
1  #Setup the start of taken routing paths
2  routestart = all.transform do |sending|
3   constraint sending[:type] == :senddone
4   constraint sending[:origin] == sending[:nodeid]
5   merging :type=> :route,:nodeid=> sending[:nodeid],:seqNo=> sending[:seqNo],:origin=> sending[:
   →origin]
6  end
7
8  # Iterate through transmissions and forwarding
9  routes = routestart.transform(:iterative) do |send, recv|
10  constraint send[:origin] == recv[:origin]
11  constraint send[:seqNo] == recv[:seqNo]
12  select send[:nodeid] == recv[:nodeid] and recv[:type] == :route and send[:type] == :senddone
13  select send[:dest] == recv[:nodeid] and send[:type] == :route and send[:type] == :receive
14  merging :type=> :route, :nodeid=> recv[:nodeid], :seqNo=> send[:seqNo], :origin=> send[:origin
   →], :dest=> send[:dest],
15 end
```

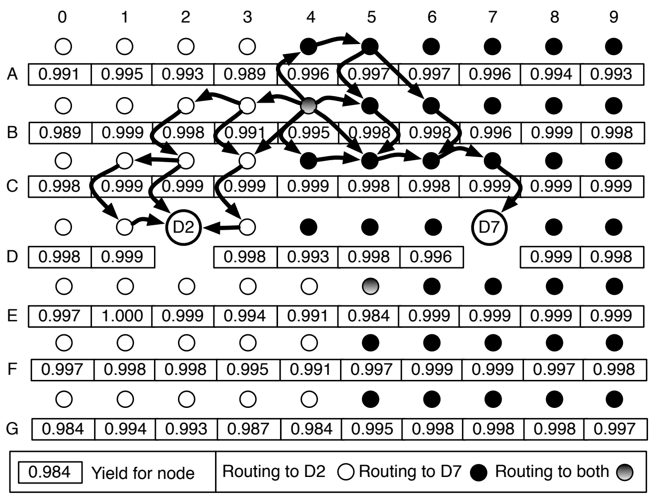
Listing 3. Routing paths analysis: First a set transformator determines all events that start the path a packet is routed on through the network (lines 2-6). Subsequently the fixed point processor iteratively builds up the taken path of the packet by associating transmissions across nodes and forwarding on nodes (lines 9-15).

is determined. First, the event must indicate a sent packet (`:senddone`, line 3) and secondly, the origin of the event must match its node identifier (line 4). The transformation function maintains the inherent information in the event, but changes the type to `:route` to mark the route start (line 5). The fixed point processor (line 9) associates a send and receive event. Only packets with the same sequence number and origin are joined (lines 10-11). Now a given event is joined with either with a new sent event on the same node (line 12) or with a receive event of a node to which a message is sent to (line 13, `sending[:dest] == receive[:nodeid]`). Finally, the events are merged into a `:route` event (line 14), which may be further processed in a further iteration.

IV. Case study

As an example for an analysis of a data gathering application, Rupeas is applied to data from simulating MultihopOscilloscope with two sink nodes in TOSSIM [2]. Some of the questions Rupeas can answer are: Which paths were data packets actually sent along? What is the average hop count? Are packets routed to both of the sinks equally? To this end, the Collection Tree Protocol (CTP) is instrumented to log sent and received packets. The simulation topology is a 70 node grid (cf. Fig. 1) including two sink nodes (0, 100). The simulation uses gain and noise models based on USC's Realistic Wireless Link Quality Model and Generator [3]. 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 and using the fixed point processor (cf. Listing 3). Rupeas allows for filtering routing paths for final destination, route selection and hop count. The main results obtained are visualized in Figure 1, where all nodes are indexed by their row (A-G) and column (0-9)



	0	1	2	3	4	5	6	7	8	9
A	0.991	0.995	0.993	0.989	0.996	0.997	0.997	0.996	0.994	0.993
B	0.989	0.999	0.998	0.991	0.995	0.998	0.998	0.996	0.999	0.998
C	0.998	0.999	0.999	0.999	0.999	0.998	0.998	0.999	0.999	0.999
D	0.998	0.999		0.998	0.993	0.998	0.996		0.999	0.998
E	0.997	1.000	0.999	0.994	0.991	0.984	0.999	0.999	0.999	0.999
F	0.997	0.998	0.998	0.995	0.991	0.997	0.999	0.999	0.997	0.998
G	0.984	0.994	0.993	0.987	0.984	0.995	0.998	0.998	0.998	0.997

Fig. 1. Visualization of Rupeas results.

position[4]: It depicts the yield of each individual node and the sink it mainly sent packets to. The overall yield is high (99.6%) and the traffic is generally routed evenly among the sinks: sink D7 receives moderately more packets (52.7%). Most nodes route their packet to a single sink, except node B4 and node E5, which send to both sinks (e. g. node B4 sends 50.4% of its packets to sink D2). The yield for those nodes, especially B4, are nevertheless high. Hop counts for the individual routes differ considerably. For node B4, packets are routed via minimally two and maximally six intermediate hops (average: 5.303). When considering average hop count per origin node, the longest average routes to sink D2 are from node G5 ($\approx$ 6 hops) and to sink D7 from node A8 ($\approx$ 7 hops). Hence, Rupeas shows that the CTP provides a high data yield for all nodes and routes traffic evenly among the two sinks in this test scenario.

[3]http://anrg.usc.edu/www/index.php/Downloads

[4]As an example, the node where the routes originate from is node B4.

V. Discussion and related work

Rupeas allows for performing common analyses of test executions, e. g. for data collection applications described above: What is the data yield from each node? What do the routing paths look like? Such analysis can be performed by Rupeas even for multi-sink systems as shown in Sec. IV. As an example, routing paths are determined by formulating a relation on send and receive events gathered in traces. Rupeas does not rely on any information about a specific application or test platform, but rather generates all routing paths by applying user-specified relations on event sets. Further analysis on these productions identifies the desired information, e. g. average routing path length to determine efficiency.

Previously proposed approaches cannot help with such analyses: 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 [8], [9]. While event based approaches analyze causal or temporal sequences, i. e. patterns, signifying a specific behavior, such global state-based approaches focus on snapshots of the distributed state, i. e. at a certain instant in time. Hence, these state-based approaches are rather targeted at identifying problems such as selection of a parent of the routing protocol, but cannot help in the analysis of taken routes. Another difference is that with assertions, the collection of information and analysis and oracles are tightly coupled. Even tighter integration is intended with Wringer [10], 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. Rupeas takes a different approach: The monitoring and information collection is decoupled allowing for running different analyses on the same monitoring data or analyses from different test platforms.

Analysis tools from other domains, e. g. wired distributed systems, typically have large requirements on the instrumentation of a system. As an example, Pip [11] is a system for automatically checking the behavior of a distributed system by specifying (or generating) expectations of program behavior and checking the executions. However the considerable requirements on instrumentation prohibit its use in resource- and instrumentation-constrained WSNs. As discussed, these limitations in instrumentation lead to different logging policies and differing communication relations on individual protocol layers. These application- and logging-specific behavior needs to be integrated in the analysis. Hence, standard methods as used in wired distributed systems cannot be employed. Analysis of wireless systems has focussed on inferring information from passively collected information, e. g. in WSNs [12], [13] and in WLAN [14], [15]; however for tests, logging information is complete, yet dependent on application and test.

VI. Summary

Rupeas is a 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 allows for integration into test and design frameworks or maintenance routines. Rupeas is an open source project (http://code.google.com/p/rupeas/).

VII. 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.

References

[1] J. Beutel, R. Lim, A. Meier, L. Thiele, C. Walser, M. Woehrle, and M. Yuecel, "Poster abstract: The FlockLab Testbed Architecture," in *Proc. 7th ACM Conf. Embedded Networked Sensor Systems (SenSys 2009)*, November 2009, pp. 415–416.

[2] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: Accurate and scalable simulation of entire TinyOS applications," in *Proc. SenSys 2003*, Nov. 2003, pp. 126–137.

[3] M. Woehrle, C. Plessl, R. Lim, J. Beutel, and L. Thiele, "EvAnT: Analysis and checking of event traces for wireless sensor networks," in *Proc. of the IEEE Intl. Conf. on Sensor Networks, Ubiquitous, and Trustworthy Computing*. IEEE, June 2008, pp. 201–208.

[4] M. Woehrle, C. Plessl, J. Beutel, and L. Thiele, "Increasing the reliability of wireless sensor networks with a distributed testing framework," in *Proc. 4th Workshop on Embedded Networked Sensors (EmNets 2007)*, June 2007, pp. 93–97.

[5] J. L. D. Chu, F. Zhao 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.

[6] M. Woehrle, C. Plessl, and L. Thiele, "Rupeas: Ruby Powered Event Analysis DSL," Computer Engineering and Networks Laboratory, ETH Zurich, Tech. Rep. 290, February 2009.

[7] M. Maifi, H. Khan, T. Abdelzaher, and K. K. Gupta, "Towards diagnostic simulation in sensor networks," in *Distributed Computing in Sensor Systems*. Springer, 2008, pp. 252–265.

[8] M. Lodder, G. Halkes, and K. Langendoen, "A global-state perspective on sensor network debugging," in *Proc. of the 5th Workshop on Hot Topics in Embedded Networked Sensors (HotEmNets 2008)*, 2008.

[9] K. Römer and M. 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] A. Tavakoli, D. Culler, P. Levis, and S. Shenker, "The case for predicate-oriented debugging of sensornets," in *Proc. of the 5th Workshop on Hot Topics in Embedded Networked Sensors (HotEmNets 2008)*, 2008.

[11] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, "Pip: detecting the unexpected in distributed systems," in *Proc. of the 3rd Symposium on Networked Systems Design & Implementation (NSDI'06)*, 2006, pp. 115–128.

[12] M. Ringwald and K. Römer, "Deployment of sensor networks: Problems and passive inspection," in *Proc. of the 5th Workshop on Intelligent Solutions in Embedded Systems (WISES '07)*, June 2007, pp. 180–193.

[13] M. Maifi, H. Khan, L. Luo, C. Huang, and T. Abdelzaher, "SNTS: Sensor network troubleshooting suite," in *Distributed Computing in Sensor Systems*. Springer, 2007, pp. 142–157.

[14] Y.-C. Cheng, J. Bellardo, P. Benkö, A. C. Snoeren, G. M. Voelker, and S. Savage, "Jigsaw: solving the puzzle of enterprise 802.11 analysis," in *SIGCOMM '06*, 2006, pp. 39–50.

[15] R. Mahajan, M. Rodrig, D. Wetherall, and J. Zahorjan, "Analyzing the MAC-level behavior of wireless networks in the wild," *SIGCOMM Comput. Commun. Rev.*, no. 4, pp. 75–86, 2006.