THEMIS: A Tool for Decentralized Monitoring Algorithms

Antoine El-Hokayem, Yliès Falcone

To cite this version:

Antoine El-Hokayem, Yliès Falcone. THEMIS: A Tool for Decentralized Monitoring Algorithms. ISSTA 2017, Jul 2017, Santa Barbara, United States. pp.125-135 10.1145/3092703.3092723 . hal-01653727

HAL Id: hal-01653727
https://hal.archives-ouvertes.fr/hal-01653727

Submitted on 1 Dec 2017

THEMIS: A Tool for Decentralized Monitoring Algorithms

Antoine El-Hokayem
Univ. Grenoble Alpes, Inria,
Laboratoire d'Informatique de Grenoble
F-38000 Grenoble, France
antoine.el-hokayem@univ-grenoble-alpes.fr

Yliès Falcone
Univ. Grenoble Alpes, Inria,
Laboratoire d'Informatique de Grenoble
F-38000 Grenoble, France
ylies.falcone@univ-grenoble-alpes.fr

ABSTRACT

THEMIS is a tool to facilitate the design, development, and analysis of decentralized monitoring algorithms; developed using Java and AspectJ. It consists of a library and command-line tools. THEMIS provides an API, data structures and measures for decentralized monitoring. These building blocks can be reused or extended to modify existing algorithms, design new more intricate algorithms, and elaborate new approaches to assess existing algorithms. We illustrate the usage of THEMIS by comparing two variants of a monitoring algorithm.

KEYWORDS

Runtime Verification, Monitoring, Tool, Java, AspectJ

1 INTRODUCTION

Runtime Verification. Runtime Verification (RV) [1, 10, 12] is a lightweight formal method which consists in verifying that a run of a system is correct with respect to a user-provided specification. The specification formalizes the expected behavior of the system. Typically, the system is considered as a blackbox that feeds events to a monitor. An event usually consists of a set of atomic propositions describing abstract operations or states in the system. A sequence of such events is referred to as a trace. When receiving a trace, the monitor will emit verdicts in a truth domain that indicate the compliance of the system to the specification. We focus on methods to monitor decentralized systems, that is, systems with multiple components having no central observation point. In decentralized systems, the monitors have a partial view of the system and need to account for communication [6] in addition to computation.

Existing Approaches. Several algorithms have been designed to monitor decentralized systems, they are detailed in [8]. They can be summarized into two different approaches. The first approach consists in monitoring by formula rewriting such as rewriting Linear Temporal Logic (LTL) [6, 13]. Typically a formula is rewritten and simplified until it is equivalent to $\top$ (true) or $\bot$ (false) at which point the algorithm terminates. The second approach [5] is concerned with consensus on the verdict with fault-tolerance. Monitors may fail to receive correct observations or communicate state with other monitors. This approach determines the necessary verdict domain needed to be able to reach a consensus. Algorithms are primarily designed to address one issue at a time and are typically experimentally evaluated by considering runtime and memory overheads. However, such algorithms are difficult to compare as they may combine multiple approaches at once. For example, algorithms that use rewriting not only exhibit variable runtime behavior due to the rewriting, but also incorporate different monitor synthesis approaches that separate the specification into multiple smaller specifications depending on the monitor. DecentMon [3, 6] was developed and extended to study the behavior of three decentralized monitoring algorithms that rely on LTL rewriting. DecentMon uses various measures to assess the algorithms both on computation and communication overhead. The measures presented are related to the delay, representing an extra time imposed by communication to generate the verdict, number and size of messages transferred across the system components and the number of progressions, representing the rewrites done to the formula. DecentMon runs the benchmarks on the three algorithms, generates the necessary synthetic traces and reports the measures. However it does not easily allow for flexibility to tune the existing algorithms, experiment with different measures, develop new variants, and it only supports LTL specifications.

2 THE THEMIS APPROACH

Methodology. THEMIS [9] is written in Java, uses AspectJ [11] and is provided as a library with a set of command-line tools. The primary goal of THEMIS is to design and analyze decentralized monitoring algorithms. It is addressed mostly for researchers to experiment, tune, and compare decentralized monitoring algorithms. To assess the behavior of an algorithm, we identify four phases (Figure 1): design, instrument, execute, and analyze. The design phase consists in elaborating a monitoring algorithm. THEMIS generalizes the monitoring steps and provides an API to describe the operations. These operations are used as building-blocks to assemble an algorithm. The instrument phase consists in the definition of measures. Measures are instrumented into THEMIS and the algorithms at run-time and operate on the API and data structures. The execute phase consists in using the THEMIS set of tools to run simulations of the monitoring algorithms and record the measures. The analyze phase consists in using the recorded measures to study, compare, and refine the algorithm. In this paper, simulations are executed on synthetic traces of programs. An instrumented program feeds THEMIS a stream of events (by implementing the interface Trace).

Design Goals. The main design goal of THEMIS is to provide a general API for decentralized monitoring. That is, to provide an environment that accounts for changes at all levels: traces, specification, monitoring logic. By doing so, we allow for new approaches implementing the API to benefit from all existing metrics and analysis. Additionally, this allows metrics to be assessed at the abstract level, for example the metric `messages sent` could be simply reused if new algorithms exchange messages. Following this goal, we also

July 2017

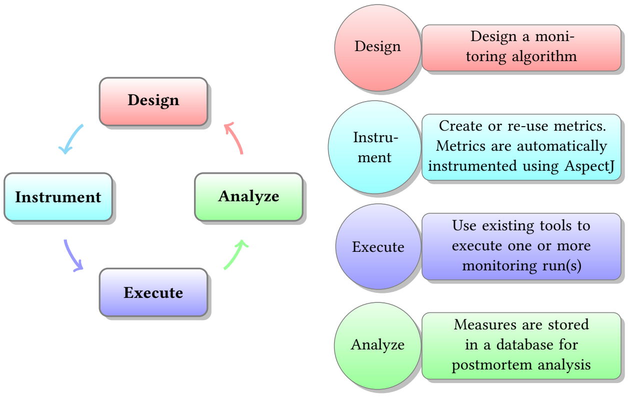

Design
Instrument
Analyze
Execute
Design
Design a monitoring algorithm
Instrument
Create or re-use metrics. Metrics are automatically instrumented using AspectJ
Execute
Use existing tools to execute one or more monitoring run(s)
Analyze
Measures are stored in a database for postmortem analysis

Figure 1: Using the THEMIS Framework

aimed that our measures be stored per run in a database. This allows for analysis and benchmarking to be reduced to querying and analysis of the database. This effectively separates the analysis from the monitoring. Third-party tools can be used to explore and analyze the data. Another important design goal is reproducibility. We wanted to minimize the effort of re-running older simulations or comparing new approaches with older ones. This is reflected with the `Experiment` command-line tool which, in short, allows users to bundle all traces, specifications and algorithms. Since metrics are designed to work at the API level and data structures, any algorithm using the same building blocks can be measured similarly without added effort. This allows for new algorithms or variants of older algorithms to be easily compared against older ones with the same data and measures. By accomplishing these two primary goals, we minimize the overhead needed to design new algorithms and study them, and let researchers focus on the algorithm and the monitoring itself. Finally, THEMIS is designed to introduce decentralized specifications [8]. That is, having different specifications for different components in the system. While some approaches [2, 6] do in effect introduce a decentralized specification, they primarily focus on presenting one global formula of the system from which they derive multiple specifications. THEMIS encompasses [2, 6] and in addition supports any decentralized specification.

3 THE THEMIS FRAMEWORK

Monitoring. We begin by explaining the basic layout of a generalized decentralized monitoring algorithm. A monitoring algorithm has two phases: setup and monitor. In the first phase, the algorithm creates and initializes the monitors, connects them to each other so they can communicate, and attaches them to components so they receive the observations generated by components. In the second phase, each monitor receives observations at a timestamp based on the component it is attached to. The monitor can then perform some computation, communicate with other monitors, abort monitoring or report a verdict. To accomplish this we use the two interfaces `MonitoringAlgorithm` and `Monitor`. In the basic use case, `MonitoringAlgorithm` is expected to provide the `setup()` method, which does the setup phase of the algorithm, and returns a map specifying monitors and their ids. A monitor has to implement the `monitor()` method for the monitoring logic, and the `reset()` method to reset its state when executing multiple runs. The method `monitor()` provides the monitor with a timestamp and a memory of observations at that timestamp based on the monitored component. The provided flow of the base `MonitoringAlgorithm` is similar to the Bulk Synchronous Parallel (BSP) [14] model. In the BSP model, all processes execute a computation phase, then, they communicate and finally synchronize. The timestamp is associated with the round number. The monitoring phase begins by setting up the monitor network. Then, for each timestamp, the observations are gathered from the trace, then all monitors execute their `monitor()` method.

Listing 1 Main Instrumentation Methods

```
public void setupRun(MonitoringAlgorithm alg);
protected void runBegin();
protected void stepBegin(int t);
protected void stepEnd(int t);
protected void stepReport(int t, ReportVerdict rep);
protected void runEnd();
```

Measuring. THEMIS uses AspectJ to record measures of a metric for a given algorithm. Writing a metric for an algorithm consists in using AspectJ's aspects to intercept the points in the execution. To simplify the task, THEMIS provides the base aspect `Instrumentation` and the classes `Measure` and `MeasureFunction`. The Instrumentation aspect already defines basic pointcuts and triggers simple methods upon reaching them, they are shown in Listing 1. When running THEMIS tools to execute the monitoring algorithms, metrics are instrumented into the code at load-time using AspectJ's Load-Time Weaving (LTW) configuration. This is found in `aop.xml`, the file configures the AspectJ agent that weaves the aspects during load-time. Thus, by pre-loading `aop.xml` measures can be enabled or disabled for a specific run.

Traces. The provided tools and algorithms use a simple format to represent components and the traces of events they receive. The components are named alphabetically starting with a (for example: a, b, c). The observations bound to the components are prefixed by the component and followed with a number starting from 0 (for example: `a0`, `a1`, `a2`, `b0`, `b1`). The trace consists of multiple files, prefixed by the trace ID and suffixed by the component name. A trace for two components a and b consists of two files: `1-a.trace` and `1-b.trace`. Each line in the file consists of an event.

Specifications. A top level specification is by default a decentralized specification. A decentralized specification is a collection of specifications. Specifications are passed to a monitoring algorithm as a `Map`, where each specification is identified by a key. Each specification must provide two attributes: an id and a class name. The id is a string name for the specification, it is used by the algorithm during the `setup` phase. The provided algorithms use the id `root` to denote the main specification. The class name is a string representing a full class name of the specification class. Listing 2 shows an example of LTL specification. It is given the name `root` and is loaded by the class `SpecLTL`. THEMIS will instantiate a `SpecLTL` object and invoke the `setLTL(String)` method with the LTL formula passed as string. THEMIS currently handles both LTL and Automata specifications and supports loading from `dot` files similar to those exported by `ltl2mon` [4].

Execution History Encoding (EHE). For the demonstration, we focus on specifications formalized using automata. The execution

Listing 2 An LTL Specification

```
<specifications>
  <specification id="root"
              class="uga.corse.themis.monitoring.SpecLTL">
    <setLTL><![CDATA[XXXX(!a0 | (b1 U G(a0 & b0 & c0)))]]></setLTL>
  </specification>
</specifications>
```

of the specification automaton, is in fact, the process of monitoring, upon running the trace, the reached state determines the verdict. In a decentralized system, a component receives only local observations, it generally does not have enough information to determine the state at a given timestamp. Typically, when sufficient information is shared between various components, it is possible to know the state reached in the automaton. The EHE is a data structure that encodes the execution of the automaton using boolean expressions and ensures strong eventual consistency in determining the state reached in the execution. Formal details are in [8].

Command-line Tools. The THEMIS framework is bundled with several tools to execute monitoring. The Run tool takes as input the name of a monitoring algorithm class, a specification file, the number of components, the length of a trace (in order to timeout), a traces directory, and one or more traces to read and simulate the run of the algorithm on the given trace and specification. Upon finishing the execution, the measures will be printed. The Experiment tool is designed to execute a set of runs packaged as an experiment. Experiments are used to define sets of parameters, traces and specifications. An experiment is effectively a folder containing all necessary files. After running a single run or an experiment, the measures are stored in a database for postmortem analysis. These can be queried, merged, or plotted easily using third-party tools.

4 THE MIGRATION ALGORITHM

The migration algorithm is a decentralized monitoring algorithm where information is passed throughout the components to eventually verify the specification. In our setup, the migration algorithm will assign a monitor per component. These monitors are strongly connected; each monitor is connected to all other monitors. The monitors are either active or inactive. Active monitors are monitors that seek to find a verdict while inactive monitors are idle, waiting for other monitors to send them their EHE. Both active and inactive monitors store the observations they receive in their memory. However, only active monitors will update the expressions in their EHE. After expressions are updated, active monitors will determine which other monitors should be sent the EHE to continue monitoring using a method getNext. In this demonstration, we use two different implementations of getNext. The first chooses the next monitor by cycling through all monitors in a round-robin fashion. The second chooses the monitor based on the earliest observation missing to evaluate the EHE [6].

Setup. Listing 3 shows the setup phase. We first make sure to convert the main specification (identified by root) to an automaton specification (line 2-3). Next, we create the monitors map, and generate as many monitors as components, giving them ids starting from zero. Each monitor is then attached to a component (line 8) to receive observations on that component. We note that in the default implementation of communication, all monitors are connected to each other, therefore there is no need to connect monitors to each other.

Listing 3 Migration Setup Phase

```
 1  protected Map<Integer, ? extends Monitor> setup() {
 2    config.getSpec().put("root",
 3        Convert.makeAutomataSpec(config.getSpec().get("root")));
 4    Map<Integer, Monitor> mons = new HashMap<Integer, Monitor>();
 5    Integer i = 0;
 6    for(Component comp : config.getComponents()) {
 7      MonMigrate mon = new MonMigrate(i);
 8      attachMonitor(comp, mon);
 9      mons.put(i, mon);
10      i++;
11    }
12    return mons;
13  }
```

Monitor. The monitoring logic of the monitor is shown in Listing 4. First, the monitor updates its memory by adding the new observations (line 3). Then, the monitor checks if it received anything and merges the received EHE. If the monitor receives anything then they become active. Upon receiving observations the monitor then updates its EHE and checks for a verdict. If a verdict has changed (line 9) and the verdict reached is a final verdict (line 11), then we report it and remove unnecessary entries in the EHE (line 13). We then determine the id of the new monitor to send the EHE to (line 15). The two implementations of the method getNext() determine the variants of Migration. If it is a different monitor id, then we must migrate, the EHE is sent to the next monitor (line 18) and the current monitor is rendered inactive (line 19).

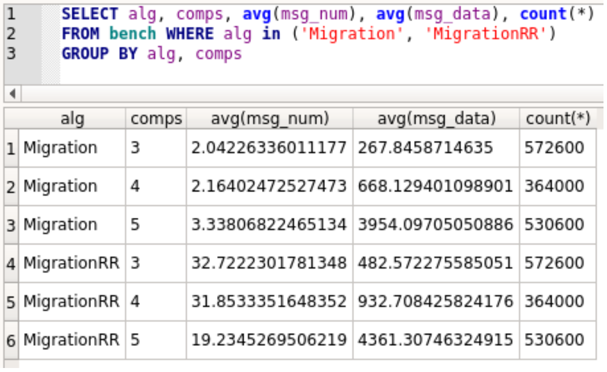
Listing 4 Migration Monitor

```
 1  public void monitor(int t, Memory<Atom> observations)
 2  throws ReportVerdict, ExceptionStopMonitoring {
 3      m.merge(observations);
 4      if(receive()) isMonitoring = true;
 5      if(isMonitoring) {
 6        if(!observations.isEmpty())
 7          autRep.tick();
 8        boolean b = autRep.update(m, -1);
 9        if(b) {
10          VerdictTimed v = autRep.scanVerdict();
11          if(v.isFinal())
12            throw new ReportVerdict(v.getVerdict(), t);
13          autRep.dropResolved();
14        }
15        int next  = getNext();
16        if(next != getID()) {
17          Representation toSend = autRep.sliceLive();
18          send(next, new RepresentationPacket(toSend));
19          isMonitoring = false;
20        }
21      }
22    }
```

Measures. To evaluate the behavior of the migration algorithm we use the communication as an example metric. We measure the number of messages sent and the size of the messages. The number of messages indicates the number of migrations performed, while the size of the messages indicates how big the EHE is. The number of messages is shown in Listing 5. We begin by adding the measure and initializing with zero (line 2). We intercept the message sending using AspectJ (line 4) and update our measure (line 5).

Listing 5 Measuring Communication

```
protected void setupRun(MonitoringAlgorithm alg) {
 addMeasure(new Measure("msg_num","Msgs",0L,Measures.addLong));
}
after(Integer to, Message m) : Commons.sendMessage(to, m) {
 update("msg_num" , 1L);
}
```



```
SELECT alg, comps, avg(msg_num), avg(msg_data), count(*)
FROM bench WHERE alg in ('Migration', 'MigrationRR')
GROUP BY alg, comps
```

	alg	comps	avg(msg_num)	avg(msg_data)	count(*)
1	Migration	3	2.04226336011177	267.8458714635	572600
2	Migration	4	2.16402472527473	668.129401098901	364000
3	Migration	5	3.33806822465134	3954.09705050886	530600
4	MigrationRR	3	32.7222301781348	482.572275585051	572600
5	MigrationRR	4	31.8533351648352	932.708425824176	364000
6	MigrationRR	5	19.2345269506219	4361.30746324915	530600

Figure 2: Example Database Querying

Analyze. We aim to compare the communication patterns of the two variants. To do so, we execute 2,934,400 runs to generate a database of the measures. We use 200 traces of 100 events per component, we associate with each component 2 observations. We vary the number of components between 3 and 5, and ensure that for each number we have at least 1,000 formulae that reference all components. Specifications are generated as random LTL formulae using `randltl` from `Spot` [7], then converted to automata using `ltl2mon` [4]. Figure 2 displays our example query on the database to retrieve the communication measures, where column alg (resp. comps, avg(msg_num), avg(msg_data), count) indicates the algorithm (resp. number of components, average number of messages, average size of messages, the number of runs). MigrationRR stands for Migration with round robin. We can see that the naive round-robin variant has both a higher number of messages and more communication. The smaller number of messages indicates that less migrations are performed overall.

5 CONCLUSION

We present `THEMIS`, a tool for designing and analyzing decentralized monitoring algorithms. `THEMIS` is an extensible framework for analyzing current algorithms, designing new ones, and experimenting with variants of existing algorithms. In addition, `THEMIS` provides common metrics for communication and computation overheads. `THEMIS` allows for re-use of both algorithms and measures so as to allow for easier comparison between algorithms. We provide an example of the methodology by applying it to the migration algorithm.

Finally, it is possible to check the tool, more examples, usage tutorial, technical documentation, and the (reproducible) experiments conducted with `THEMIS` on its website [9].

REFERENCES

[1] Ezio Bartocci, Yliès Falcone, Borzoo Bonakdarpour, Christian Colombo, Normann Decker, Klaus Havelund, Yogi Joshi, Felix Klaedtke, Reed Milewicz, Giles Reger, Grigore Rosu, Julien Signoles, Daniel Thoma, Eugen Zalinescu, and Yi Zhang. 2017. First international Competition on Runtime Verification: rules, benchmarks, tools, and final results of CRV 2014. *International Journal on Software Tools for Technology Transfer* (2017), 1–40. DOI:http://dx.doi.org/10.1007/s10009-017-0454-5

[2] David A. Basin, Felix Klaedtke, and Eugen Zalinescu. 2015. Failure-aware Runtime Verification of Distributed Systems. In *35th IARCS Annual Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2015 (LIPIcs)*, Prahladh Harsha and G. Ramalingam (Eds.), Vol. 45. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 590–603.

[3] Andreas Bauer and Yliès Falcone. 2016. Decentralised LTL monitoring. *Formal Methods in System Design* 48, 1-2 (2016), 46–93.

[4] Andreas Bauer, Martin Leucker, and Christian Schallhart. 2011. Runtime Verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.* 20, 4 (2011), 14.

[5] Borzoo Bonakdarpour, Pierre Fraigniaud, Sergio Rajsbaum, David A. Rosenblueth, and Corentin Travers. 2016. Decentralized Asynchronous Crash-Resilient Runtime Verification. In *27th International Conference on Concurrency Theory (CONCUR 2016) (LIPIcs)*, Josée Desharnais and Radha Jagadeesan (Eds.), Vol. 59. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 16:1–16:15.

[6] Christian Colombo and Yliès Falcone. 2016. Organising LTL monitors over distributed systems with a global clock. *Formal Methods in System Design* 49, 1-2 (2016), 109–158.

[7] Alexandre Duret-Lutz. 2013. Manipulating LTL formulas using Spot 1.0. In *Proceedings of the 11th International Symposium on Automated Technology for Verification and Analysis (ATVA'13) (Lecture Notes in Computer Science)*, Vol. 8172. Springer, 442–445.

[8] Antoine El-Hokayem and Yliès Falcone. 2017. Monitoring Decentralized Specifications. In *Proceedings of 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'17), Santa Barbara, CA, USA, July 2017.* DOI:http://dx.doi.org/https://doi.org/10.1145/3092703.3098224

[9] Antoine El-Hokayem and Yliès Falcone. 2017. THEMIS Website. (2017). https://gitlab.inria.fr/monitoring/themis.

[10] Yliès Falcone, Klaus Havelund, and Giles Reger. 2013. A Tutorial on Runtime Verification. In *Engineering Dependable Software Systems*, Manfred Broy, Doron a. Peled, and Georg Kalus (Eds.). NATO science for peace and security series, Vol. 34. IOS press, 141–175.

[11] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An Overview of AspectJ. In *ECOOP 2001 - Object-Oriented Programming, 15th European Conference (Lecture Notes in Computer Science)*, Jørgen Lindskov Knudsen (Ed.), Vol. 2072. Springer, 327–353.

[12] Martin Leucker and Christian Schallhart. 2009. A brief account of runtime verification. *J. Log. Algebr. Program.* 78, 5 (2009), 293–303.

[13] Grigore Rosu and Klaus Havelund. 2005. Rewriting-Based Techniques for Runtime Verification. *Autom. Softw. Eng.* 12, 2 (2005), 151–197.

[14] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (Aug. 1990), 103–111.