Towards Modeling Monitoring of Smart Traffic Services in a Large-scale Distributed System

Andreea Buga and Sorana Tania Nemeș
Christian Doppler Laboratory for Client-Centric Cloud Computing, Johannes Kepler University, Softwarepark 35, Hagenberg im Mühlkreis, Austria

Keywords: Formal Modeling, Monitoring Service, Traffic Surveillance, Abstract State Machines.

Abstract: Smart traffic solutions have become an important component of today's cities, due to their aim of improving the quality of the life of inhabitants and reducing the time spent in transportation. They are deployed across large distributed systems and require a robust infrastructure. Their complex structure has been addressed numerous times in practice, but rarely in a formal manner. We propose in this paper a formal modeling approach for monitoring traffic systems and identifying possible failures of traffic sensors. Ensuring a safe and robust deployment and execution of services implies having a clear view on the system status, which is analysed by the monitoring framework. Our work focuses on availability aspects and makes use of the Abstract State Machines modeling technique for specifying the solution. The framework is defined as an Abstract State Machine agent and simulated in the ASMETA tool.

1 INTRODUCTION

Design and development of large-scale distributed systems (LDSs) have been widely researched in the last years due to the continuously growing storage and processing demands in information systems' area. We address in our research project the aspects related to monitoring nodes composing an LDS, with a practical example of traffic monitoring services. We consider monitoring to represent acquiring data about the status of the nodes and other information offered by their internal sensors.

The main goal of our work is to construct a formal specification for the monitoring services, through using algorithms specific to distributed systems. Building a formal model evolves from natural-language use cases and user stories and focuses on capturing correctly functional and non-functional requirements. We started from examining traffic systems and elaborating the requirements, from which we built the formal model.

In order to ensure that such systems deliver reliable and correct data to the users within an admissible period of time, we need to observe possible failures that might occur either at node level or during communication. Interpreting information and assessing the state of the system and also of the traffic is, therefore, essential. The solution we propose is a formal model for a monitoring framework deployed along the LDS, with components replicated at node-level. We use Abstract State Machine(ASM) method for elaborating the specifications.

In comparison with previous proposals in the area of monitoring smart traffic systems, we are now focusing on identifying failures of sensor nodes and assigning a robust diagnosis through collaboration of several monitors.

The remainder of the paper is structured as follows. In section II we define the problem statement and the motivation. Section III familiarizes the reader with the necessary concepts and introduces the ground model for the monitors, that we refine to a pseudo-code-like/AsmetaL characterization, and simulate. Section IV explores previous meaningful work carried out in the area, after which the paper is concluded in Section V.

2 PROBLEM STATEMENT

LDSs have become a necessary paradigm for ensuring the needed capabilities of nowadays services. Existing traffic surveillance solutions for smart cities benefit of the evolution of distributed systems methods capture a huge amount of data. Cloud computing is a successful example of an LDS, which can offer a

Buga, A. and Nemes, S.
Towards Modeling Monitoring of Smart Traffic Services in a Large-scale Distributed System.
DOI: 10.5220/0006303704830490
In *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*, pages 455-462
ISBN: 978-989-758-243-1

robust infrastructure for traffic monitoring.

We aim to define a model for the smart surveillance of traffic and identify possible issues that might occur inside the system. We propose a framework for monitoring the availability of the sensors, named throughout the paper as nodes. We are interested also in collecting the data offered by the nodes and interpret it to understand the current situation of the traffic. The monitoring framework ensures the availability of the smart traffic services.

Monitoring is essential for understanding the performance and evolution of the system. Traditionally, it implies collecting data from running services and assessing system performance in terms of service availability, failures and anomalies. The heterogeneity and distribution of the sensors of a traffic system over a wide geographical area has introduced a high complexity for the monitoring solutions. The use of a formal model covers the system analysis and design phases of software development and leads to a proposal which can be verified for its intended properties.

Model-driven engineering allows the stakeholders to contribute at defining specific concepts and entities. Natural language requirements, Unified Modeling Language (UML) use cases, agile user stories are captured in models who are part of the software development process. In the design phase functional and non-functional properties are proposed and validated. Spotting errors later in the development process leads to higher costs for software projects.

The specification needs to encompass the behavior of the monitors and abstract away from complex details. We favored the use of the ASM method in front of other modeling techniques as, UML or Business Process Model and Notation (BPMN), due to its ability to model multi-agent systems and to easily refine specifications by replacing an action through multiple parallel actions. In comparison with UML, ASMs provide a more rigorous abstraction, that allows verification through model checking.

3 SYSTEM OVERVIEW

Traffic surveillance solutions consist of a large number of sensors deployed and communicating across an LDS. The proposed monitoring framework is part of an architecture model concerned with coordinating numerous heterogeneous components. The architecture of the whole system is expressed as an abstract machine model as depicted in Fig. 1. The monitoring component is closely related to the execution layer from where it extracts information and the adaptation layer, which uses information from it to bring the system to a proper state. The diagnosis established by the monitor focuses on three main aspects: failure detection, assessment of availability and diagnosis of network problems (failure of the communication processes).

The ASM relies also on local storage for saving important events and data. Monitoring information is saved in terms of low- and high-level metrics in the data storage, while meaningful operations (adaptation events, identification of problems) are stored in the event database. A meta storage is used for saving additional information as for instance functions to aggregate low-level metrics.

Robustness of the proposal is achieved by employing redundant monitors that can take over the tasks in case of the misbehavior of running elements. Therefore, each traffic sensor is assigned a set of monitors to assess its status. The evaluation is carried out in a collaborative way. When one of the monitors exhibits a random behavior, it is stopped by the middleware and replaced.

Moreover, the interaction of the monitoring and adaptation layers enables the system to perform reconfiguration plans whenever any of the sensors faces a problem. The monitoring framework submits the collected data to the adapter whenever a problem occurs. Afterwards, a plan to restore the system to a normal working state is proposed and the monitors perform a new evaluation that can indicate if the adaptation processes have been efficient.

3.1 Background on ASM

Our research focused on elaborating formal models for monitoring the smart traffic solutions in terms of ASMs, which allow capturing the requirements in abstract specifications that can further be implemented.

The method offers system descriptions that can be easily understood by the clients, as well as developers. ASMs have already been used in industrial projects, in proofs of correctness of programming languages (Börger and Stark, 2003) and in modeling client-cloud interaction (Arcaini et al., 2016).

One of the main artefacts of ASMs is the ground model, which reflects system's requirements. It is advanced, through incremental refinements, to a written specification, that can be simulated and validated before deployment.

Basic ASMs contain transition rules expressed as **if** *Condition* **then** *Updates*, where the *Condition* is an arbitrary predicate logic formula and the *Updates* are defined as a set of assignments $f(t_1, ..., t_n) := t$. For an update to be carried out successfully, consistency must be ensured, meaning that for each location only

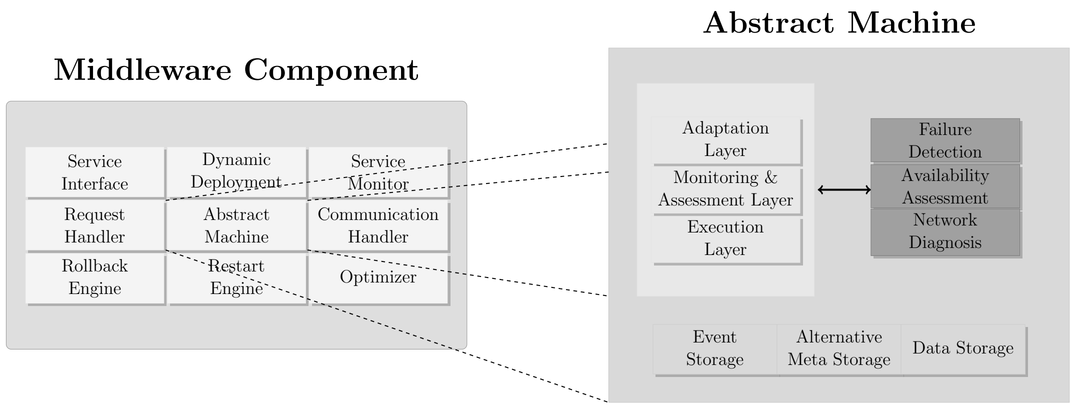

Figure 1: Structure of the Middleware and its internal Abstract Machine.

one value has to be assigned (Börger and Stark, 2003).

LDSs usually rely on multiagents ASMs, which better reflect their modular organization. Each agent executes its own rules in parallel on its local states. Each monitor component is represented in our program as an ASM agent. The system we describe is asynchronous, and hence it cannot ensure a global state, but rather a stable local view of each monitor.

Constants in ASMs are expressed as static functions, while variables are expressed as dynamic functions. A classification of the dynamic functions depends on the agent who is allowed to operate on them. For instance, *controlled* functions can be modified only by the agent and read by the environment, while *monitored* ones can be written only by the environment and read internally by the agent. It is worth mentioning, that from the perspective of an agent, the environment is represented by the other agents. *Shared* functions can be modified by both the agent and the environment. A more exhaustive description of the ASM method and functions is given by (Börger and Stark, 2003).

We make use of ASMETA framework, a toolset tailored for defining and simulating ASM models. We are interested at this step in defining, simulating and observing their behavior at runtime. (Gargantini et al., 2008).

3.2 System Requirements

As mentioned in the previous subsection, ASMs mediate the translation of requirements to specifications. At design phase, we had into consideration the following list of requirements:

- [R. 1.]Each node of the system is assigned a set of monitors;
- [R. 2.]Monitoring is carried out at different levels of the system and follows and hierarchical structure;
- [R. 3.]After monitoring is started, availability of a node is checked through heartbeat requests;
- [R. 4.]The framework must detect problems of a node based on the data collected;
- [R. 5.]The monitor must disseminate the information locally and gossip about detected problems;
- [R. 6.]Data collected are processed and temporarily stored;
- [R. 7.]The monitoring processes run continuously in the background of execution of normal services;
- [R. 8.]The trust in the assessment of a monitor decreases with the number of incorrect diagnoses it made;
- [R. 9.]When a monitor is considered untrustworthy, the middleware will stop the activity of the monitor;

3.3 ASM Ground Model

As illustrated in Fig. 2, we encompassed all the requirements previously listed in the control state diagram of the local monitor assigned to a node. We complete the diagram with sequences of the ASM rules elaborated in AsmetaL language, which is part of the ASMETA framework for modeling various services and prototyping (Riccobene and Scandurra, 2014).

In equivalence with the control state diagram, the monitoring agent can be in one of the following states:

```
enum domain State = {INACTIVE | IDLE | ACTIVE |
WAIT_RESPONSE | COLLECT_DATA | RETRIEVE_INFO |
ASSIGN_DIAGNOSIS | LOG_DATA | REPORT_PROBLEM};
```

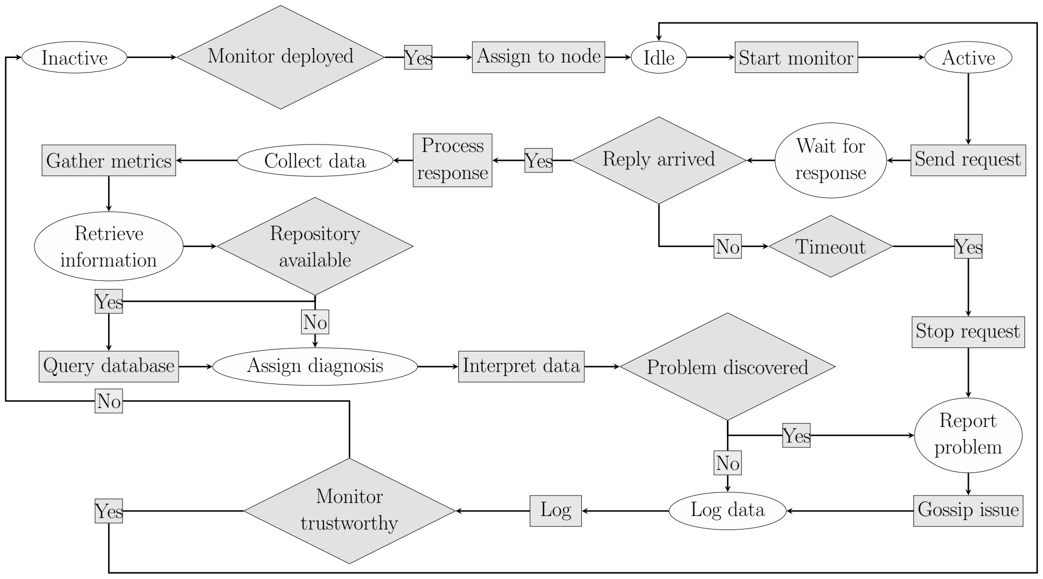

Figure 2: ASM ground model for the monitor.

In the initial phase of the execution, the monitor is in *INACTIVE* state. It awaits for its deployment in the system, which is carried out by the middleware and modeled as a monitored boolean function:

```
monitored monitorDeployed : Monitor -> Boolean
```

After its deployment in the system, the monitor has to be assigned to a node. This process is currently randomly binding a monitor to a sensor. In order to ensure fairness, we aim to refine this process so that a new monitor is distributed to the node with the smallest number of monitors. The monitors designated to a node are stored in the *nodeMonitors* list.

```
rule r_assignToNode($mon in Monitor) =
 if (assigned($mon)) then
  skip
 else
  choose $n in Node with true do
   par
    assigned($mon) := true
    nodeMonitors($n) := append (nodeMonitors($n),
          $mon)
   endpar
 endif
```

Once the monitor is appointed to a node, it reaches the *IDLE* state, from where the monitoring process starts and it moves to the *ACTIVE* state. Sending a request implies creating a *Heartbeat* instance and sending it to the node to discover latency and availability.

```
rule r_sendRequest ($mon in Monitor) =
 extend Heartbeat with $h do
  seq
   heartbeatStatus($h) := SUBMITTED
   heartbeats($mon) := append(heartbeats($mon),
        $h)
  endseq
```

Subsequently to submitting the request, the monitor advances to the *WAIT_RESPONSE* state. The guard *Reply arrived* verifies if a response to the heartbeat is acknowledged. If the monitor records a response, it inspects it and moves to the *COLLECT_DATA* state. The measurements are stored in a set of $< key, value >$ pairs, where the key is the unique string identifier of the metric and the value is filled by the monitor. The pairs are stored in the local repository at every monitoring cycle. The monitor gathers all the available data from the node and moves to the *RETRIEVE_INFORMATION* state. At this point it checks if it can access the local repository and if so, it queries the database. In case it is not possible to obtain information from the local storage, the monitor moves to the *ASSIGN_DIAGNOSIS* state directly.

The rule for interpreting the data is in charge with processing the collected metrics and evaluating the current state of the sensor node. When a problem is identified, as well as in the case the heartbeat reply takes longer than an accepted maximum delay, the monitor moves to the *REPORT_PROBLEM* state. From there gossip communication is triggered and the other monitors assigned to the current node submit their own assessment. The diagnosis set by the majority of the nodes is the one taken into consideration. We left the gossip rule abstract for the moment, as there are various gossip protocols that can be used by different systems. After establishing a diagnosis by consulting with the other counterparts, the decision is locally logged and the confidence degree of the monitor is calculated after the following formula:

$$confDegree(mon) = confDegree(mon) - c(mon) \cdot penalty \cdot \frac{|diagnoses| - |similar\_diagnoses|}{numberOfDiagnoses} \quad (1)$$

where the initial confidence degree of a monitor is:

```
function confDegree($m in Monitor) = 100
```

and

$$c(mon) = \begin{cases} 0, & \text{if the diagnosis was correct} \\ 1, & \text{otherwise.} \end{cases}$$

As it can be noticed, the confidence degree is a strictly monotonically decreasing function in order to prevent the case the confidence degree is increased by false positive diagnosis. A diagnosis is considered correct if it matches the decision adopted by the majority of the monitors. In case of a false diagnosis, the confidence degree of a monitor is influenced by the marginality of its assessment. For instance, if the diagnosis is shared by more counterparts, then the confidence degree decreases with a smaller value. The value of the *penalty* constant depends on how critical a mistaken diagnosis is for the system and is defined at the initialization of the system. If the confidence degree value of a monitor is above a set threshold, it starts a new monitoring cycle. Otherwise, it will move to the *INACTIVE* state and wait for the middleware to take an action.

```
 function isMonitorTrustworthy($mon in Monitor) =
  if (gt(conf($mon), minConf)) then
     true
  else
   false
  endif
```

Monitors are also components of the LDS and they can be affected by failures and unavailability problems just like any other node of the system. Using the confidence measure for evaluating the correctness of a monitor we aim to mitigate the failures of the monitoring solution itself.

Monitoring processes produce a high volume of data, which needs to be processed for assessing the evaluation. In order to avoid data cluttering and loading the local storage, we proposed a data degradation approach. Hence, the information which is older than a specific date are considered irrelevant and are removed from the repository.

3.4 Relevant Monitoring Metrics

Basic monitoring data does not offer a complete view over the system. Having a delayed response does not give extra information about probable network congestion or overloaded sensors. Unsuccessful data logging does not reveal problems related to the storage, and so on. Combining basic data into higher-level metrics provides a better understanding of the state of the node, helps in establishing a better diagnosis and finding proper adaptation measures.

The monitoring framework detects abnormal runtime situations of sensors of the smart traffic system. The tasks executed by such nodes have to be taken over by another suitable components. We describe a composed metric characterizing the *process transfer effort (PTE)*, which is of interest in case the adaptation component decides that a node needs to be replaced with another candidate. The metric comprises information about the candidate node (availability, performance and reliability), which need to be defined in their turn from other basic data as available storage, memory consumption and network bandwidth. Due to privacy issues and different security policies, it needs to be established if a new node can take over the data of the faulty component. If a task cannot be moved due to legislation issues, the value of the PTE for the new node is set to ∞.

```
**if** !canBeLegallyTransferred(i, t) **then**
    $pte(i,t) = \infty$
**end if**
```

For a node *i*, among *k* eligible candidates for a task *t* we propose the following metric:

$$pte(i,t) = \frac{1}{bdwidth(t)} \cdot \frac{perf(i)}{max(perf(k))} \cdot \frac{reliab(i)}{100}, \quad (2)$$

where *bdwith(t)* refers to the bandwidth available for submitting task *t*, *perf(i)* represents the performance of the node *i*, *max(perf(k))* is the maximum value of performance from all the available *k* candidates. *Reliab(i)* refers to the reliability of the node candidate *i*. The PTE gives information about the best node candidate to move a traffic surveillance task to. It evaluates a node according to the performance *perf* relatively to other nodes, its absolute reliability and the absolute bandwidth of the communication established for submitting the task. The lower the PTE, the most suitable the competitor node.

Node work capacity describes an overview of the percentage of available resources. This metric is important for understanding the behavior of the node when sending new requests. High consumption of resources leads to undesired delays which sometimes might be interpreted as being caused by network communication issues. Resources refer to storage, CPU and memory of a sensor. A high usage value of any of the resources leads to a significantly smaller capacity for taking new tasks. For a node i, the work capacity is equal to the percentage of resources left available after withdrawing the current efforts:

$$workCapacity(i) = \frac{100}{3} \cdot (3 - \frac{cpuUsage(i)}{100} - \frac{memoryUsage(i)}{100} - \frac{storageUsage(i)}{100}) \quad (3)$$

As mentioned before, the latency of a node i to respond to a request r is introduced either by its overloading or by network infrastructure problems. Knowing that the bandwidth is small, reduces the suspicions that there might be problems with the node and the other way around. For sensor networks, the connection between nodes is susceptible to different issues, reason for which it is important to include it in the assessment of the status of the node.

$$delay(r,i) = (100 - workCapacity(i)) \cdot \frac{1}{bdwidth(r)} \quad (4)$$

We defined the metrics also inside the rule designated to interpret the data. It works on the assumption that the structure of the measurement list is known. We temporarily made use of simple comparisons for evaluating the state of the system but we aim to advance this function so that it can make use of more complex statistic analysis methods.

```
rule r_interpretData ($mon in Monitor) =
 seq
  workCapacity($mon) := 100.0 * (itor(3) - second
       (at(dataCollected($mon), 1n)) /100.0 -
       second(at(dataCollected($mon), 3n)) /100.0
       - second(at(dataCollected($mon), 4n))
       /100.0) / 3.0
  delay($mon) := (100.0 - workCapacity($mon)) /
       second(at(dataCollected($mon), 2n))
  if ((delay($mon) > 2.0) or (workCapacity($mon)
       < 30.0) or (second(at(dataCollected($mon),
       6n)) < 40.0)) then
   par
    diagnosis ($mon) := "Critical"
    problemDiscovered($mon) := true
   endpar
  else
   diagnosis ($mon) := "Normal functioning"
  endif
  assessedDiagnosis ($mon) := append (
       assessedDiagnosis($mon), diagnosis($mon))
 endseq
```

The initial set of data to be collected by the monitor contains the following metrics: latency expressed in milliseconds needed by a node to reply to a request, CPU, memory and storage usage represented in the percentage of used resource, bandwidth expressed in Mb/s, the cost of a node to perform a task and its performance expressed as a percentage of the actual efficiency in report to the capabilities of a node. The list of metrics is yet to be completed with other relevant functions which provide insights on the status of the traffic sensors and their execution.

3.5 Simulation of the Model

We verified through simulation if the monitor model passes through the states as expressed in the control state diagram and if monitors are able to identify possible problems of a sensor node. We used AsmetaS[1] tool to simulate the execution of the model. The simulation consisted in following a specific scenario for execution. Different configurations trigger a different workflow. For compactness, we took into consideration a monitor and a smaller set of metrics for comparison and analysis, and left aside the collaborative diagnosis. The set of metrics and the values of the measurement are initialized in the function *dataCollected*. We present in Listing 1 an excerpt of the simulation of our model and we can notice that after passing through the state of *ASSIGN_DIAGNOSIS* the monitor correctly evaluates the node as being in a *Critical* execution mode. The evaluation is carried out by analysing the rule for interpreting the data, which specifies that in the case of a performance value below 40.0, the monitor assesses that the node is in a *Critical* state.

```
function dataCollected($mon in Monitor) =
 switch ($mon)
  case mon1 : [("Latency", 5.0), ("CPU Usage",
       84.0), ("Bandwidth", 150.0), ("Memory Usage
       ", 50.0), ("Storage Usage", 44.0), ("Cost",
        4.0), ("Performance", 20.0)]
 endswitch
```

4 RELATED WORK

LDSs, especially grids and clouds, are in the focus of researchers due to their high computing capacities and resources. Modeling properties of such systems imposes, nonetheless, some restrictions as researchers need to abstract away from several details that might impede proper verification.

Modeling cloud systems has been proposed by the MODA-Cloud project in order to obtain self-adaptive multi-cloud applications. It relies on CloudML language, an extension of UML, for modeling the runtime processes and specifying the data, QoS models and monitoring operation rules (Bergmayr et al.,

[1]http://asmeta.sourceforge.net/download/asmetas.html

Listing 1: Excerpt of simulating the monitor agent.

```
INITIAL STATE:Monitor={mon1}
Node={n1,n2}
Insert a boolean constant for monitorDeployed(mon1
    ): true
<State 0 (monitored)>
monitorDeployed(mon1)=true
</State 0 (monitored)>
<State 1 (controlled)>
Monitor={mon1}
Node={n1,n2}
assigned(mon1)=true
nodeMonitors(n2)=[mon1]
state(mon1)=IDLE
</State 1 (controlled)>
<State 2 (controlled)>
Monitor={mon1}
Node={n1,n2}
assigned(mon1)=true
nodeMonitors(n2)=[mon1]
state(mon1)=ACTIVE
</State 2 (controlled)>
<State 3 (controlled)>
Heartbeat={Heartbeat!1}
Monitor={mon1}
Node={n1,n2}
assigned(mon1)=true
heartbeatStatus(Heartbeat!1)=SUBMITTED
heartbeats(mon1)=[Heartbeat!1]
nodeMonitors(n2)=[mon1]
state(mon1)=WAIT_RESPONSE
</State 3 (controlled)>
Insert a boolean constant for replyArrived: true
<State 3 (monitored)>
replyArrived=true
</State 3 (monitored)>
<State 4 (controlled)>
Heartbeat={Heartbeat!1}
Monitor={mon1}
Node={n1,n2}
assigned(mon1)=true
heartbeatStatus(Heartbeat!1)=SUCCESSFUL
heartbeats(mon1)=[Heartbeat!1]
nodeMonitors(n2)=[mon1]
state(mon1)=COLLECT_DATA
</State 4 (controlled)>
<State 5 (controlled)>
Heartbeat={Heartbeat!1}
Monitor={mon1}
Node={n1,n2}
assigned(mon1)=true
heartbeatStatus(Heartbeat!1)=SUCCESSFUL
heartbeats(mon1)=[Heartbeat!1]
nodeMonitors(n2)=[mon1]
state(mon1)=RETRIEVE_INFO
</State 5 (controlled)>
Insert a boolean constant for repositoryAvailable(
    mon1):
false
<State 5 (monitored)>
repositoryAvailable(mon1)=false
</State 5 (monitored)>
<State 6 (controlled)>
Heartbeat={Heartbeat!1}
Monitor={mon1}
Node={n1,n2}
assigned(mon1)=true
heartbeatStatus(Heartbeat!1)=SUCCESSFUL
heartbeats(mon1)=[Heartbeat!1]
nodeMonitors(n2)=[mon1]
state(mon1)=ASSIGN_DIAGNOSIS
</State 6 (controlled)>
<State 7 (controlled)>
Heartbeat={Heartbeat!1}
Monitor={mon1}
Node={n1,n2}
assessedDiagnosis(mon1)=["Critical"]
assigned(mon1)=true
delay(mon1)=0.39555555555555555
diagnosis(mon1)="Critical"
heartbeatStatus(Heartbeat!1)=SUCCESSFUL
heartbeats(mon1)=[Heartbeat!1]
nodeMonitors(n2)=[mon1]
outMess="Problem discovered"
problemDiscovered(mon1)=true
state(mon1)=REPORT_PROBLEM
workCapacity(mon1)=40.66666666666667
</State 7 (controlled)>
................................................
```

2015). The ASM method we used allows a more rigurous elaboration of specification in comparison with UML.

Traffic control systems have been formally modeled in terms of Petri Nets from the point of view of safety (List and Cetin, 2004). We shift the attention towards formally modeling failure detection inside a traffic system and focus on availability of the traffic sensors.

A solution for modeling monitoring smart traffic solutions using smart agent and emphasizing on self-* properties has been proposed by (Haesevoets et al., 2009). The authors present the organization of the system and formally express the roles of the agents and their capabilities.

The necessity of modeling and validating critical systems has been reported also by (Glässer et al., 2008), which focuses on capturing the security of aviation processes with the aid of ASMs and probabilistic modeling techniques reported also by (Arcaini et al., 2015), where authors describe a medical system in a formal approach through the aid of AsmetaL.

Previous approaches of using ASMs in expressing grid services propose the description of job management and service execution in (A. Bianchi, 2011), work that was further extended by (Bianchi et al., 2013). Specification of grids in terms of ASMs has been proposed also by (Nemeth and Sunderam, 2002), with a focus in underlining differences between grids and normal distributed systems. Our work can be considered an extension of these projects with a focus on monitoring components which are responsible of service execution and detecting possible failures that can occur at node level.

5 CONCLUSIONS

The use of formal models before starting the development of a software system leads to more robust solutions. Being able to design a model according to the natural language specifications is very well supported by the ASM technique through its ground models and transition rules. We presented in the current paper a formal approach for defining a monitoring framework for a smart traffic system deployed in an LDS. We started from the requirements and elaborated the ground model and specific transition rules. One can easily infer from the structure the complete workflow of the system.

The work stays at the ground of future validation and verification of the specifications, which will help the practitioners to construct robust and reliable smart traffic solutions and their intrinsic monitoring services. As a future work, we intend to further refine the proposed model and investigate it through validation and model-checking until we obtain a prototype. By these means, issues faced by such real-time systems, like complexity and failures, can be overcome.

REFERENCES

A. Bianchi, L. Manelli, S. P. (2011). A Distributed Abstract State Machine for Grid Systems: A Preliminary Study. In P. Ivnyi, B. T., editor, *Proceedings of the Second International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering*. Civil-Comp Press.

Arcaini, P., Bonfanti, S., Gargantini, A., Mashkoor, A., and Riccobene, E. (2015). Formal validation and verification of a medical software critical component. In *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 80–89.

Arcaini, P., Holom, R.-M., and Riccobene, E. (2016). Asm-based formal design of an adaptivity component for a cloud system. *Form. Asp. Comput.*, 28(4):567–595.

Bergmayr, A., Rossini, A., Ferry, N., Horn, G., Orue-Echevarria, L., Solberg, A., and Wimmer, M. (2015). The Evolution of CloudML and its Manifestations. In *Proceedings of the 3rd International Workshop on Model-Driven Engineering on and for the Cloud (CloudMDE)*, pages 1–6, Ottawa, Canada.

Bianchi, A., Manelli, L., and Pizzutilo, S. (2013). An ASM-based Model for Grid Job Management. *Informatica (Slovenia)*, 37(3):295–306.

Börger, E. and Stark, R. F. (2003). *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Gargantini, A., Riccobene, E., and Scandurra, P. (2008). A metamodel-based language and a simulation engine for Abstract State Machines. *j-jucs*, 14(12):1949–1983.

Glässer, U., Rastkar, S., and Vajihollahi, M. (2008). *Modeling and Validation of Aviation Security*, pages 337–355. Springer Berlin Heidelberg, Berlin, Heidelberg.

Haesevoets, R., Weyns, D., Holvoet, T., and Joosen, W. (2009). A formal model for self-adaptive and self-healing organizations. In *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 116–125.

List, G. F. and Cetin, M. (2004). Modeling traffic signal control using Petri nets. *IEEE Transactions on Intelligent Transportation Systems*, 5(3):177–187.

Nemeth, Z. N. and Sunderam, V. (2002). A formal framework for defining grid systems. In *Cluster Computing and the Grid, 2002. 2nd IEEE/ACM International Symposium on*, pages 202–202.

Riccobene, E. and Scandurra, P. (2014). A formal framework for service modeling and prototyping. *Formal Aspects of Computing*, 26(6):1077–1113.