Towards Modeling Monitoring Services for Large-Scale Distributed Systems with Abstract State Machines

Andreea Buga and Sorana Tania Nemeș

Christian Doppler Laboratory for Client-Centric Cloud Computing,
Johannes Kepler University of Linz,
{andreea.buga,t.nemes}@cdcc.faw.jku.at

Abstract. The evolution of Large-Scale Distributed Systems is strongly associated with the development of solutions for smart cities. They consist of a large-number of sensors, processing centers and services deployed in a wide geographical area. Due to their complexity and heterogeneity, such systems face a high-level of uncertainty and the failure of one node can affect the availability of the whole solution. Monitoring services collect data about the state of components and elaborate a diagnosis, aiming to increase the reliability of the system. This paper proposes an Abstract State Machine model to capture the properties and behavior of monitoring services addressing system failures. The method encompasses the translation of the requirements of the system to ground models. We discuss the formal solution with respect to the problem domain and execute a simulation of the model. We discuss the suitability of the method for distributed systems and compare it with other modeling approaches.

Keywords: Formal Modeling, Abstract State Machines, Large-Scale Distributed Systems, Monitoring, Smart City

1 Introduction

Large-Scale Distributed Systems (LDS) aggregate computing resources through a wide area network. Such systems offer scalability and transparency of resources and compose services for building various applications. Their complexity introduces many challenges like heterogeneity, node or communication failures. Recovery and high availability of the system require reliable monitoring.

One of the trends for developing a sustainable future is supported by smart cities. They encompass applications for enhancing transportation, energy usage, waste disposal. Sensors, data centers and computing resources collaborate to process data and provide services to end devices. These services face failures and availability issues of LDS. Monitors play a key role in detecting issues and providing data for adaptation plans to bring the system to a normal execution mode. Monitoring processes are complemented with adaptation processes, which respond to the existing problems with various restoration plans. The main contribution of the paper consists in analyzing and validating correct behavior of monitors, whose accuracy enhances the robustness of the system.

The goal of this paper is to integrate the formal modeling capabilities of the Abstract State Machines (ASMs) for defining a monitoring solution for LDS. We motivate the choice of ASMs by comparing them with other formal methods with respect to their suitability for distributed systems. We present the requirements of the system from the perspective of a smart city application and propose a structure for the monitoring framework. Requirements are translated to a control state ASM. The use of formal methods for defining a solution helps in understanding possible flaws of the model before deployment [10].

The remainder of the paper is structured as follows. Section 2 captures the problem domain and the research objectives of the paper. Essential concepts related to the ASM formal method are presented in Section 3. Section 4 describes the structure of the monitoring framework and is continued by its formal specification and validation in Section 5. Related work is discussed in Section 6, after which conclusions are drawn in Section 7.

2 Smart City Application Case Study

The evolution of distributed systems, Internet of Things (IoT) and network capabilities played an important role in the adoption of ubiquitous solutions for smart cities. Widely distributed sensors for traffic, pollution and environment continuously collect data that are integrated in various applications. The aim is to sustainably develop cities and improve the quality of life of the the inhabitants.

One of the main areas of interest is provisioning of medical services. Asthma is a chronic inflammatory disease manifested by airflow obstruction, coughing and/or chest tightness. The condition is directly affected by the environment and by the behavioral patterns of the patient. The benefits of a smart city application empowers patients to take informed decisions and prevent severe asthma attacks. In a smart city network, air quality sensors provide data related to the percentage of dust particles and pollutants, while meteorological data supply humidity and temperature values. Information about traffic is important for avoiding crowded areas and also indicate the pollution level. Such sensors are distributed in an LDS and data they provide can be integrated with activity patterns extracted from smart gadgets for building a knowledge base. Hosseini et al. [11] proposed an architecture for employing wireless environmental sensors within a smartwatch application that assesses the asthma risk level.

System nodes refer to sensors and services, which are offered by various cloud providers (Amazon, Microsoft Azure, etc.). Node problems are propagated to the whole system, making it hard to identify the source. We emphasize the role of the monitors for ensuring availability of the system and propose a formal model for it. The proposal closely follows the subsequent research questions and objectives.

Research Question 1. *Can formal methods capture properties of LDS monitors? How does applying formal methods to distributed systems differ from modeling traditional applications?*

We analyze existing formal methods and establish the best option given the characteristics of distributed systems. The choice of the ASM technique is jus-

tified in Section 3 together with the definition of specific control structures and properties that can be specified using this approach.

Research Question 2. *How can unavailability issues of smart city applications be tackled by the monitoring solution?*
In Section 4 we present the main requirements of the system and how the proposed monitoring model addresses them. We emphasize the unavailability issues and discuss the novelty of our approach.

Research Question 3. *How does the ASM model reflect the properties of the monitoring framework?*
We define the structure of the monitoring solution, capture the workflow in terms of control state diagram and discuss the important transitions of the system. We also declare states and rules with the aid of AsmetaL language, which reflect the behavior of the monitors. Section 5 discusses in more details these aspects.

3 Abstract State Machine Theory

While traditional software development processes integrated formal methods easier, the evolution of agile methods, distributed systems and novel business models introduced more challenges. Kossak and Mashkoor [12] propose an evaluation of the existing formal methods considering modeling criteria, supported development phases, tool support, social aspects and industrial applicability.

Given the characteristics of the system described in Section 2, we are interested in adopting the technique that supports modeling properties of distributed systems like concurrency and non-determinism. The expressiveness of the model is important due the heterogeneous nature of the target system. According to [12] the best candidates for these aspects are ASMs and TLA+. By further considering the assistance of the model through the software development process, its coherence and the scalability in industrial applications [12] we adopted the ASM method. The Unified Modeling Language (UML) is widely adopted in software engineering. However, it is considered imprecise and attempts to improve its operational semantics led to extended mathematical specifications [8].

Petri Nets have been widely used for modeling distributed systems. In [4], Börger illustrates specific distributed scenarios for assessing the capabilities of both ASMs and Petri Nets. The paper does not aim to exhaustively assess the performances of the methods, but to highlight their abstraction capabilities and graphical complexity. The ASM remarks itself as being able to capture various concepts in simpler graphical representations.

ASMs rely on the concept of evolving algebras proposed by Yuri Gurevich in [9]. Their proposal was motivated by their power to improve Turing machines with semantic capabilities. The ASM method allows a straightforward transition from natural-language requirements to ground model and control state diagrams, which can be easier formalized. An ASM machine M is represented as a tuple $M = (\Sigma, S_0, R, R_0)$, where Σ is the signature (the set of all functions), S_0 is the set of initial states of Σ, R is the set of rule declarations, R_0 is the main rule of the machine.

The specification of an ASM consists of a finite set of *transition rules* of the type: **if** *Condition* **then** *Updates* [3], where an *Update* consists of a finite set of assignments $f(t_1, ..., t_n) := t$. As ASMs allow synchronous parallelism execution, two machines might try to change a location with two different values, triggering an inconsistency. In this case the execution throws an error.

Rules consist of different control structures that reflect parallelism (`par`), sequentiality (`seq`), causality (`if...then`) and inclusion to different domains (`in`). With the `forall` expression, a machine can enforce concurrent execution of a rule R for every element x satisfying a condition φ: `forall` x `with` φ `do` R. Non-determinism is expressed through the `choose` rule: `choose` x `with` φ `do` R.

Definition 1. *A* control state ASM *is an ASM following the structure of the rules illustrated in Fig. 1: any control state i verifies at most one true guard, $cond_k$, triggering, thus, $rule_k$ and moving from state i to state s_k. In case no guard is fulfilled, the machine does not perform any action.*

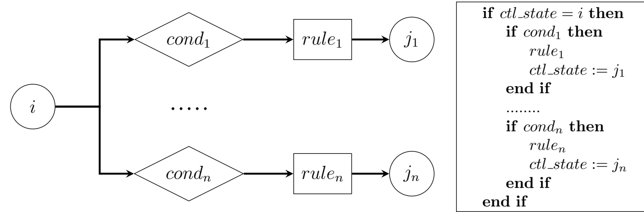

Fig. 1. Structure of a Control State ASM

Functions in ASMs are classified according to permissions on different operations. Static functions refer specifically to constants, while dynamic functions can be updated during execution. *Controlled* functions are written only by the machine, while *monitored* ones are written by the environment and read by the machine. Both the machine and its environment can update *shared* functions.

4 System Overview

This paper describes the monitoring component for an LDS, which is responsible to identify failures and unavailability of constituent nodes. The description of the system starts from the presentation of requirements and is completed by an architectural model, which emphasizes robustness achieved through redundancy.

4.1 Requirements of the Monitoring Framework

Req. 1. Monitoring processes will observe each node of the LDS. In order to avoid single points of failure, a set of monitors is assigned to every node.

Req. 2. Before starting data collection, the monitor submits a request to verify node availability. If no answer is received, the node is considered unavailable.
Req. 3. Data collected by the monitor is used for detecting unavailability problems and failures.
Req. 4. A monitor that detects a problem must disseminate it locally to other monitors assigned to the same node and carry out a collaborative evaluation.
Req. 5. Monitoring specific data and events are temporarily logged in a local storage from where they can be retrieved for analysis processes.
Req. 6. Monitoring processes run continuously in background of the execution.
Req. 7. Each monitor is characterized by a trustworthiness level, based on its performance. Bad assessment of data indicates a lower trustworthiness.
Req. 8. Monitoring data are also used for system adaptation and evaluation of reconfiguration solutions.

4.2 Organization of the Monitoring Framework

Fig. 2 illustrates the architecture of the LDS system for a smart city application. The organization comprises three parts: the client side where different users request services from providers, the provider side where sensors are deployed and an internal abstract machine for monitoring and adaptation processes. The interaction of the clients with the providers is based on a solution defined by [5], where the client-cloud interaction middleware processes the requests and ensures the delivery of services to the end user.

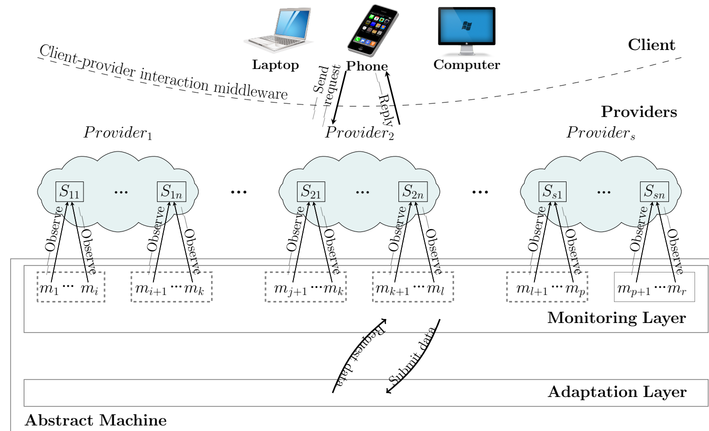

Fig. 2. Architecture of the LDS system

We assume that sensors are deployed among various providers. Sensor S_{pi} of a provider P is assigned a set of monitors Monitors $(S_{pi}) = \{m_{i1}, ..., m_{ik}\}$.

The monitors observe the node by carrying out specific processes: checking availability, collecting raw data, building higher-level metrics, interpreting data and logging. In order to reduce the communication overhead, monitors interact only when a problem is detected and a collaborative decision is required.

Monitoring components indicate abnormal situations together with corresponding data to the adaptation layer, where a case based repository is consulted and an action plan is proposed. After the deployment of the plan the adapters request data from the monitors in order to check the efficiency of their actions.

5 Formal Specification

5.1 Control state ASMs

The model contains ASM monitor agents, each carrying out its own execution according to the the requirements mentioned in Section 4.1. Fig. 3 displays the control state ASM ground model of the monitor agent. The monitor is initialized in the *Inactive* state. If the monitor is deployed by the middleware, then it can be assigned to a node. From there, the agent moves to the *Active* state from where monitoring specific processes start.

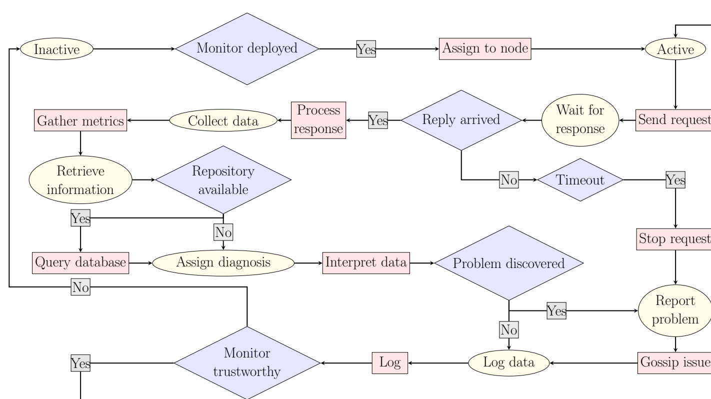

Fig. 3. Control ASM for the monitor agent

The monitor sends a request to the node after which it moves to the *Wait for response* state. The guard *Reply arrived* is verified and if an answer to the request is acknowledged, the monitor processes it by calculating the latency and moves further to the *Collect data* state. If no reply is recorded, the monitor verifies if the request has exceeded the maximum allowed delay (*Timeout* guard). In this case it stops the current request and moves to the *Report problem* state. If the *Timeout* guard is false, the agent remains in the *Wait for response* state.

In the *Collect data* state, the monitor gathers raw data from the node (CPU usage, memory usage, available storage, number of executing tasks). It moves afterwards to the *Retrieve information* state. If the guard *Repository available* is verified, the logs are queried. The monitor moves to the *Assign diagnosis state*, where data are interpreted. If the guard *Problem discovered* holds then the monitor moves to the *Report problem* state, otherwise it moves to the *Log data* state. From *Report problem* state, the monitor communicates the detected problem to other monitors assigned to the same node and moves to the *Log data* state. Information are then saved in the local repository.

Listing 5.1. AsmetaL specification of the monitor program

```
rule r_MonitorProgram =
  par
    if (monitor_state(self) = ACTIVE) then
      par
            r_SendRequest [self]
                monitor_state(self) := WAIT_FOR_RESPONSE
          endpar
    endif
        if (monitor_state(self) = WAIT_FOR_RESPONSE) then
      if (heartbeat_response_arrived(last(heartbeats(self)))) then
        if (heartbeat_timeout(last(heartbeats(self)))) then
          par
                    r_StopRequest []
                        monitor_state(self) := REPORT_PROBLEM
                  endpar
                else
                  par
                        r_ProcessResponse[]
                        monitor_state(self) := COLLECT_DATA
                  endpar
                endif
          endif
        endif
        if (monitor_state(self) = COLLECT_DATA) then
          par
                r_GatherMetrics []
                monitor_state(self) := RETRIEVE_INFO
          endpar
        endif
        if (monitor_state(self) = RETRIEVE_INFO) then
                seq
                  if (isRepositoryAvailable) then
                    r_QueryDb []
                  endif
                  monitor_state(self) := ASSIGN_DIAGNOSIS
                endseq
        endif
        if (monitor_state(self) = ASSIGN_DIAGNOSIS) then
          seq
            r_InterpretData []
                if (isProblemDiscovered(self)) then
                  monitor_state(self) := REPORT_PROBLEM
                else
                  monitor_state(self) := LOG_DATA
                endif
          endseq
        endif
        if (monitor_state(self) = REPORT_PROBLEM) then
          par
                r_GossipIssue []
                monitor_state(self) := LOG_DATA
          endpar
        endif
        if (monitor_state(self) = LOG_DATA) then
          par
                r_Log []
                if (isMonitorTrustworthy(self)) then
                  monitor_state(self) := ACTIVE
                else
                  monitor_state(self) := INACTIVE
                endif
          endpar
        endif
  endpar
```

At the end of the monitoring cycle, the trustworthiness of the monitor is calculated and if the *Monitor trustworthy* guard holds, a new cycle starts. Otherwise, the monitor moves to the *Inactive* state from where it needs to be reinitialized by the middleware. We, thus, avoid having faulty monitors in the system.

5.2 AsmetaL Specification

ASMETA [1]is a toolset for simulating and validating ASM models written in the AsmetaL language, which capture control structures and functions. The *Monitor* domain is part of the Agent universe and it behaves as an ASM machine, having its own states and transitions. Monitor state is expressed as a controlled function which is updated by the agent itself. Monitors assigned to a node are expressed as a sequence, each storing a sequence of Hearbeat requests sent to the node. The function *isProblemDiscovered* is left abstract. *isMonitorTrustworthy* function is calculated at the end of a monitoring cycle. The calculation of heartbeat timeout is a derived function combining a monitored value and a controlled function. The signature of domains and functions of the *Monitor* agent are important for the representation of the control state ASM from Section 5.1 in Listing 5.1 [2].

5.3 Validation of the Model

Currently, validation deals only with the separate processes for each agent. It checks the workflow and the transitions from different states. The model was validated with AsmetaV tool, which allows the creation of scenarios with the aid of the Avalla language described by [7]. By validation we discover possible flaws in the design of the ASM model.

For validation we created an instance of the *Node* domain, which is assigned three *Monitor* agents. We checked how various inputs affect the control flow of the monitors and if the rules and states of the agent match the control state ASM ground model from Fig. 1 as displayed in Listing 5.2. In a future step of the validation process we plan to analyze the interaction between monitor agents and the function to update the confidence degree of a monitor.

6 Related Work

Formal methods have distinguished themselves through the ability to capture mathematical properties in software specification. LDS systems introduce a higher complexity and heterogeneity that needs to be handled. We consulted the area of formal methods and chose the ASM technique proposed by [3].

Modeling LDS has been addressed in several cloud and grid related projects. CloudML, an extension of the UML language for expressing cloud specific processes, has been proposed by the MODACloud project for specifying adaptable Quality of Service (QoS) models, monitoring operation rules and data [1].

[1] http://asmeta.sourceforge.net/

[2] The complete specification is available at `http://cdcc.faw.jku.at/staff/abuga/emmsad.rar`

Listing 5.2. Example on an AsmetaV scenario

```
scenario Monitor1
set assigned_monitors(node_1) := [monitor_1, monitor_2, monitor_3];
set heartbeat(monitor_1) := heartbeat_1;
set heartbeat(monitor_2) := heartbeat_2;
set heartbeat(monitor_3) := heartbeat_3;
step
check monitor_state(monitor_1) = WAIT_FOR_RESPONSE and monitor_state(monitor_2) =
    WAIT_FOR_RESPONSE and monitor_state(monitor_3) = WAIT_FOR_RESPONSE;
set heartbeat_response_arrived(heartbeat_1) := true;
set heartbeat_response_arrived(heartbeat_2) := false;
set heartbeat_response_arrived(heartbeat_3) := true;
set heartbeat_latency(heartbeat_1) := 5;
set heartbeat_latency(heartbeat_3) := 15;
step
check monitor_state(monitor_1) = COLLECT_DATA and monitor_state(monitor_2) =
    WAIT_FOR_RESPONSE and monitor_state(monitor_3) =REPORT_PROBLEM;
set heartbeat_response_arrived(heartbeat_2) := true;
set heartbeat_latency(heartbeat_2) := 20;
step
check monitor_state(monitor_1) = RETRIEVE_INFO and monitor_state(monitor_2) =
    REPORT_PROBLEM and monitor_state(monitor_3) = LOG_DATA;
set isRepositoryAvailable := false;
set isMonitorTrustworthy(monitor_3) := true;
step
check monitor_state(monitor_1) = ASSIGN_DIAGNOSIS and monitor_state(monitor_2) =
    LOG_DATA and monitor_state(monitor_3) = ACTIVE;
set isProblemDiscovered(monitor_1) := true;
set isMonitorTrustworthy(monitor_2) := true;
step
check monitor_state(monitor_1) = REPORT_PROBLEM and monitor_state(monitor_2) = ACTIVE
     and monitor_state(monitor_3) = WAIT_FOR_RESPONSE;
```

Formal modeling was also used for building models for grid services and processes. The ASM technique contributed to the description of the job management and service execution in [2]. Specification of grids in terms of ASMs have been proposed also by [14], where Németh and Sunderam focused on expressing differences between grid and traditional distributed systems.

ASMs have been also proposed for realization of web service composition. In [13], Ma et al. introduced the notion of Abstract State Services and showed an use case for a cloud service for flight booking. Service composition and orchestration in terms of ASMs have been researched by [6].

7 Conclusions

Formal methods ensure reliable software solutions. LDS introduce a high complexity in the system and building formal models for them is still a challenging task. We discussed in this paper the aspects of monitoring LDS, proposed a set of requirements and translated them to an ASM model. The choice of the ASM technique was justified by comparing it with other available formal methods.

The current model is limited to a set of states and rules that capture the workflow of the monitors. Timing related constraints, which are essential for LDS, could not be expressed. However, the focus is on ensuring the correctness of the monitoring behavior and improving the overall robustness of the system.

As a future work, we will improve the formal model to capture finer-level details. We plan to achieve loose-coupling by employing ASM modules for different functionality of the monitoring framework. In order to ensure the correctness of the solution we will perform verification with the aid of AsmetaSMV tool.

References

1. A. Bergmayr, A. Rossini, N. Ferry, G. Horn, L. Orue-Echevarria, A. Solberg, and M. Wimmer. 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, September 2015.
2. Alessandro Bianchi, Luciano Manelli, and Sebastiano Pizzutilo. An ASM-based model for grid job management. *Informatica (Slovenia)*, 37(3):295–306, 2013.
3. E. Börger and Robert F. Stark. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.
4. Egon Börger. Modeling distributed algorithms by abstract state machines compared to petri nets. In *Proceedings of the 5th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z - Volume 9675*, ABZ 2016, pages 3–34, New York, NY, USA, 2016. Springer-Verlag New York, Inc.
5. Károly Bósa, Roxana-Maria Holom, and Mircea Boris Vleju. A formal model of client-cloud interaction. In *Correct Software in Web Applications and Web Services*, pages 83–144. 2015.
6. Davide Brugali, Luca Gherardi, Elvinia Riccobene, and Patrizia Scandurra. Coordinated execution of heterogeneous service-oriented components by abstract state machines. In *Formal Aspects of Component Software - 8th International Symposium, FACS 2011, Oslo, Norway, September 14-16, 2011, Revised Selected Papers*, pages 331–349, 2011.
7. Alessandro Carioni, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A scenario-based validation language for asms. In *Proceedings of the 1st International Conference on Abstract State Machines, B and Z*, ABZ '08, pages 71–84, Berlin, Heidelberg, 2008. Springer-Verlag.
8. Zamira Daw and Rance Cleaveland. An extensible formal semantics for UML activity diagrams. *CoRR*, abs/1604.02386, 2016.
9. Yuri Gurevich. Specification and validation methods. chapter Evolving Algebras 1993: Lipari Guide, pages 9–36. Oxford University Press, Inc., New York, NY, USA, 1995.
10. C. M. Holloway. Why engineers should consider formal methods. In *16th DASC. AIAA/IEEE Digital Avionics Systems Conference. Reflections to the Future. Proceedings*, volume 1, pages 1.3–16–22 vol.1, Oct 1997.
11. A. Hosseini, C. M. Buonocore, S. Hashemzadeh, H. Hojaiji, H. Kalantarian, C. Sideris, A. A. T. Bui, C. E. King, and M. Sarrafzadeh. Hipaa compliant wireless sensing smartwatch application for the self-management of pediatric asthma. In *2016 IEEE 13th International Conference on Wearable and Implantable Body Sensor Networks (BSN)*, pages 49–54, June 2016.
12. Felix Kossak and Atif Mashkoor. *How to Select the Suitable Formal Method foran Industrial Application: A Survey*, pages 213–228. Springer International Publishing, Cham, 2016.
13. H. Ma, K. D. Schewe, and Q. Wang. An abstract model for service provision, search and composition. In *2009 IEEE Asia-Pacific Services Computing Conference (APSCC)*, pages 95–102, Dec 2009.
14. Zsolt N. Németh and Vaidy Sunderam. A formal framework for defining grid systems. *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 0:202, 2002.