Formalizing Monitoring Processes for Large-Scale Distributed Systems Using Abstract State Machines

Andreea Buga(✉) and Sorana Tania Nemeș

Christian Doppler Laboratory for Client-Centric Cloud Computing,
Johannes Kepler University of Linz, Software Park 35, 4232 Hagenberg, Austria
{andreea.buga,t.nemes}@cdcc.faw.jku.at

Abstract. Large-Scale Distributed Systems are characterized by high complexity and heterogeneity, which might lead to unexpected failures. The role of a robust monitoring framework is to gather low-level data and assess the status of the components of the system. The framework collaborates with adapters for ensuring steady recovery plans and improving the availability of services. Monitors, as part of the system, are also affected by unavailability or random failures. In order to increase the reliability of the solution we verify the trustworthiness of the monitors and emphasize the need of redundancy. This paper introduces a formal approach for modeling and verifying a monitoring solution for Large-Scale Distributed Systems. We formalize the behavior of the monitors with the aid of Abstract State Machines and employ the ASMETA toolset for simulating and analyzing properties of the model. The tool also supports the verification process by translating a simplified version of the model to an NuSMV specification, on top of which model checking can be applied. Properties of the model are expressed with the aid of computation tree logic.

Keywords: Formal modeling · Abstract State Machines · Monitoring Failure detection · Model validation · Computation Tree Logic

1 Introduction

Service-oriented architecture (SOA) permits orchestrating resources of various providers with the aim of delivering highly available and effective services to the end user. The development of such systems relies on techniques and algorithms specific to Large-Scale Distributed Systems (LDS). Problems encountered by any component lead to bigger failures, from which the system needs to recover.

Traditionally, monitoring refers to collecting specific data from components of the system and interpreting them in order to detect possible issues. Complemented with a robust adaptation framework, monitors ensure that the system meets its requirements and performs according to the promises expressed in terms of service-level agreement (SLA).

A. Cerone and M. Roveri (Eds.): SEFM 2017 Workshops, LNCS 10729, pp. 153–167, 2018.
https://doi.org/10.1007/978-3-319-74781-1_11

Monitors are components of the LDS and are also faced with unavailability or misbehavior. False reports on problems of the nodes trigger unnecessary adaptation plans, while the impossibility to correctly detect an issue leads to lower performance of the system or even its complete failure. The goal of the monitoring service we propose is to precisely detect unavailability and crash failures of system nodes.

The current paper addresses the accuracy of the monitoring processes and adopts a formal approach for modeling their correct behavior. We impose redundancy for monitoring processes and introduce a measure for the accuracy of the monitors, referred throughout the paper as confidence degree. We specified the model for the monitoring framework with the aid of Abstract State Machines (ASM). In this sense, we engaged the methodology proposed by Arcaini et al. [4] for simulating, validating and verifying ASM models.

The remainder of the paper is organized as follows. Section 2 presents relevant work in the area. In Sect. 3 we define the architecture of the system and discuss desired behavior, which is translated into a formal specification in Sect. 4. Section 4 also contains a brief presentation of ASM specific concepts. Verification of system properties is carried out in Sect. 5. Limitations of the approach are discussed in Sect. 6, after which conclusions are drawn in Sect. 7.

2 Related Work

Lattice framework [13] proposed for the evaluation of cloud federations uses different components for executing monitoring processes. It addresses data collection, encapsulation and communication and serves as a guideline for the monitor behavior proposed by our ground model. Our work approaches the problem from the formal point of view, while Lattice discusses implementation details.

mOSAIC [21] relies on SOA and provides an Application Programming Interface (API) for communication. Every LDS component is specified in terms of resource requirements (storage, computing and communication, and budget). Brokering is done via SLA contracts at both component and application level and it is handled by the Resource Broker. S-Cube proposes also monitoring and adaptation services from the perspective of SLA violation [14]. While the focus of mOSAIC and S-Cube is on the delivery of promised SLAs, our work details the monitoring unavailability and crash failures.

ASM formal method has been used for specifying and verifying different aspects of LDS. Ma et al. introduced the notion of Abstract State Services based on ASMs and described it for a flight booking over a cloud service case study [16], while Bolis et al. proposed a formal approach for testing the conformance of web applications using ASM method [6].

In [1], Arcaini et al. propose an ASM model for analyzing Monitor-Analyze-Plan-Execute over a shared Knowledge (MAPE-K) loops of self-adapting systems that follow a decentralized architecture. Flexibility and robustness to silent node failures of the specification are validated and verified with the aid of the ASMETA toolset. The monitoring part of the MAPE-K loops was, however, not included. Arcaini et al. also addressed the specification and verification of the adaptation component for cloud systems [4].

Grid systems were also analyzed in terms of ASMs with respect to job and execution management by [5]. Differences between traditional distributed systems and grids were presented in an ASM formal description by [20]. The previously mentioned work on ASM specifications for distributed systems is representative for our understanding of distributed ASMs and their elaboration.

In our earlier work, we discussed the requirements of the monitoring solution for cloud-enabled Large-Scale Distributed Systems (CELDS) with respect to two areas of high interest for the development of smart cities [11,12]. In [11] we presented the requirements of the monitoring services for a traffic system and translated them to an ASM ground model that has been simulated. The model has been discussed in correlation with a healthcare system deployed in CELDS and his previous versions have been validated in [10,12]. The current paper extends this work and focuses on the verification of the ASM model. If the desired properties are ensured, the model can be further expanded to an actual prototype.

3 System Overview

LDS are composed of resources and services offered by various providers. One of the successful business models relying on algorithms and principles of LDS is cloud computing. While single providers cover the basic computing requirements, the increasing amount of data to be processed led to interconnecting clouds. The services are heterogeneous and their internal structure is unknown to the monitors. However, they allow the collection of a specific set of metrics, relevant for the assessment of their status. The current work has been extended on the frame of the monitoring services for CELDS, whose foundation has been presented in [17].

CELDS compose resources and services from different providers to respond to the needs of the clients. For instance, when there is a peek of requests, resources are asked from other providers. The system is a black box for the user, who is interested only in the quality of the services he requests. The execution layer, whose model and definition we take from [9], needs to be continuously monitored for flaws that might affect the availability or reliability. The role of the monitors is to detect and correctly assess faulty situations and submit this information further for system adaptation. The monitoring and adaptation components communicate intensively in order to detect and resolve problems occurring at the execution layer.

3.1 System Architecture

CELDS consist of different tiers as shown in Fig. 1. From the client-side, a large number of devices can send requests to providers. All the requests and replies are handled by a client-provider middleware formalized by [9]. Each provider,

$Provider_i$, of the system consists of a set of nodes $\{N_{i1}, N_{in}\}$, whose resources are composed and offered as services to users. Nodes can refer to a processing or a storage unit. Every node is assigned a set of monitors $\{m_1, m_i\}$ that evaluate its status. The monitoring layer communicates with the adaptation component by providing meaningful information for reconfiguration plans, which bring the system into a normal working state. The adaptation processes have been previously defined and formalized in [18,19].

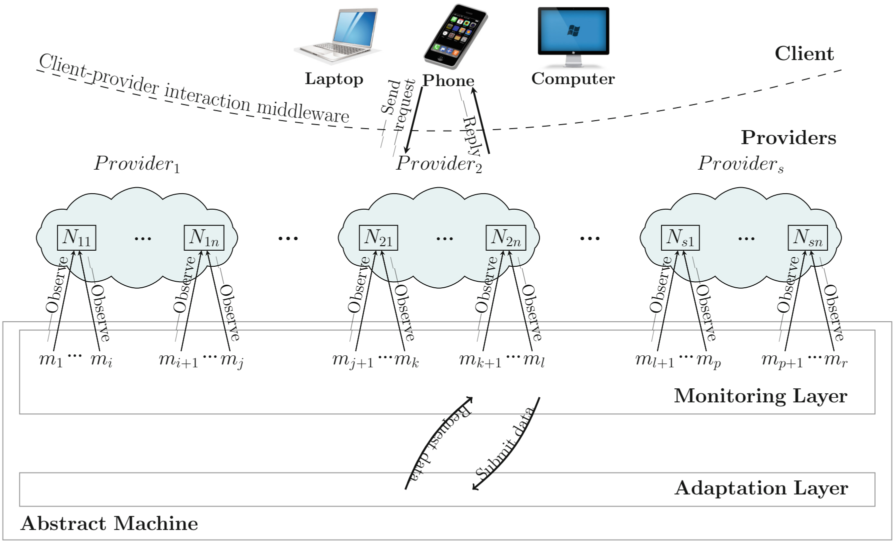


Fig. 1. Architecture of the system

3.2 Structure of the Monitoring Component

The monitoring layer continuously runs in the background of the execution layer and collects data related to each node. Monitors face also own issues. They might be unresponsive or submit random information, rather than the correct one. We introduce the notion of confidence degree based on their relative accuracy. The more imprecise evaluations a monitor does, the lower its confidence. The accuracy of a monitor is calculated as the deviation of the diagnosis it provides in comparison with the diagnosis voted by the majority of the monitors. If the confidence degree of one monitor falls below a certain threshold, it is deactivated. We assume that this process is correctly performed by the middleware and do not model this aspect.

Assessment of one monitor is not sufficiently reliable as data unavailability does not necessarily imply node failure. It can also indicate a problem of the monitor itself or of the communication link. We, then, need more monitors to contribute to the assessment. We assume that each node is observed by at least three monitors, which can collaborate and that the assignment is executed by the middleware. The minimum number of monitors represents the minimum number of participants in a voting quorum. We introduce the notion of leader of the monitors assigned to a node, which coordinates collective diagnoses whenever an issue is reported by one of the them. As shown in Fig. 2, all the monitors observe the node. $Monitor_j$ discovers a problem and reports it to the leader, which afterwards requests data from all the monitors associated to the node.

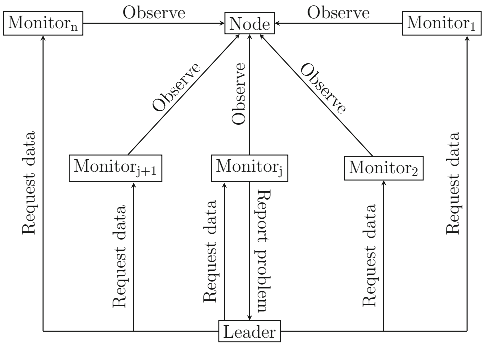


Fig. 2. View on the monitoring set assigned to a node

The leader is a different agent type, which settles a diagnosis based on the information received from all the monitors. The decision is taken based on a voting method, where each monitor inputs its own evaluation with a weight equal to its confidence degree. At the end, the diagnosis preferred by the most trustworthy majority is chosen and each of the monitor recalculates its confidence degree as follows. If the monitor inputs the same diagnosis as the one calculated by the leader, its confidence degree value does not modify. Otherwise, a penalty factor is applied. The number of similar diagnoses also contributes to the recalculation of the confidence degree. The larger the number of equivalent diagnoses to the one given by the monitor, the lower the decrease. It is, thus, considered that assessments shared by a larger number of monitors are more likely to be correct. Equation 1 shows the formula used to recalculate the confidence degree. The penalty factor is defined at initialization, depending on how critical the system is. The number of similar diagnoses represents the number of monitors who submitted the same assessment as $monitor_i$, while the number of diagnoses represents the total number of monitors who submitted their assessment.

$$conf\_degree(i) = conf\_degree(i) - \frac{|diagnoses| - |similar\_diagnoses(i)|}{|diagnoses|} \cdot *penalty\_factor,$$
$$where\ i \in Monitors(n), n \in Nodes \tag{1}$$

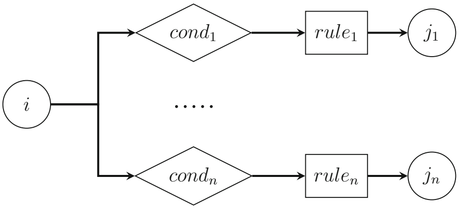
4 Formal Specification of the System

4.1 Background on ASM Theory

ASM is a formal method, which enhances the notion of Finite State Machine (FSM) with the possibility to express data structures for the *in* and *out* states connected by a transition. An ASM machine M is defined 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.

A model consists of a finite set of transition rules of type: **if** *Condition* **then** *Updates*, where the *Condition* is an arbitrary predicate logic formula and the *Updates* is defined as a set of assignments to a *location* represented as a function f having a list of dynamic parameters $t_1, \ldots, t_n$: $f(t_1, \ldots, t_n) := t$. The method permits expressing synchronous parallelism, in which an update might attempt to assign distinct values to a location, thus leading to inconsistent updates. The following definition supported by Fig. 3 has been given by [7].

Definition 1. *A* control state ASM *is an ASM following the structure of the rules illustrated in Fig. 3: 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.*



```
if $ctl\_state = i$ then
    if $cond_1$ then
        $rule_1$
        $ctl\_state := j_1$
    end if
    ........
    if $cond_n$ then
        $rule_n$
        $ctl\_state := j_n$
    end if
end if
```

Fig. 3. Structure of a control state ASM

Rules of an ASM indicate control structures emphasizing parallelism (`par`), sequentiality (`seq`) and causality (`if...then`). With the `forall` expression, a machine can enforce concurrent execution of a rule R for every element x that satisfies a condition φ: `forall` x `with` φ `do` R. Non-determinism is expressed through the `choose` rule: `choose` x `with` φ `do` R.

Kossak and Mashkoor compared different formal models in [15] with respect to their expressiveness, easiness to use, integration in the software development process, and learning curve. ASM and Temporal Logic of Actions (TLA+) methods proved a good suitability for distributed systems. Petri Nets were not included in the study, but a comparison with the ASM method on concrete examples was carried out by Börger in [8], where Petri Nets proved to generate more complex and hard to follow specifications.

4.2 Overall Workflow of Model Specification and Analysis

Elaboration, validation and verification of the model follow a set of steps depicted in Fig. 4. System requirements are first captured by an ASM model, which can be easily defined with the AsmetaL language. Transformation of the monitoring processes requirements to ASM ground models has been previously expressed in [11,12]. The ASMETA model can be further simulated or validated by building specific scenarios as detailed in [10]. The tool permits also automatic review of the model for properties like conciseness or for design issues using the AsmetaMA adviser, and the AsmetaSMV tool generates a NuSMV model, which can be verified against desired properties.

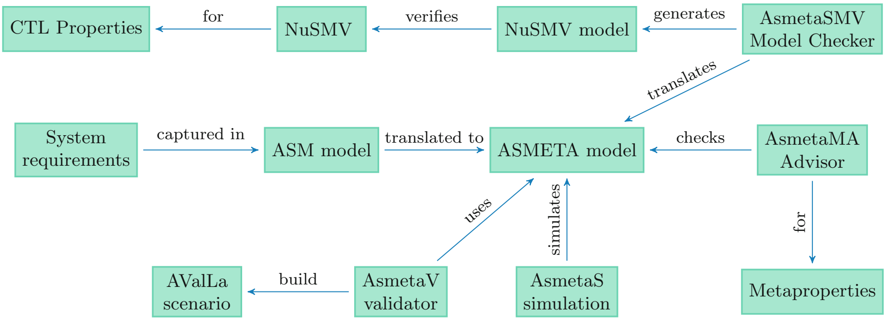


Fig. 4. Overall workflow of the modeling and verification processes

4.3 ASM Specification of the Monitoring Solution

The ASM specification of the monitoring solution closely matches the description from Sect. 3.2 and its structure is depicted in Fig. 5. The model consists of a middleware agent, responsible for initializing the system and administration operations (assignment of monitors, deactivation of untrustworthy components). The model contains two main modules, one for the monitor and one for the leader. We also used the *CTLlibrary* offered by the ASMETA toolset for verification. CELDS nodes are defined as elements of a domain and they contain a few functions relevant for the monitoring processes. We left abstract their formal specification and focused on the monitoring part. For the verification part, the model has been reduced to one agent and the functions have been simplified so that they contain primitive, finite data types.

The monitor module corresponds to the ground model depicted in Fig. 6 and relies on the description of [10,11]. Each monitor is initialized by the middleware agent in the *Inactive* state. As soon as it is deployed, it is assigned to a node and moves to the *Active* state. From this state it sends a ping request (referred further in the paper as heartbeat request) to verify if the node is available and moves to the *Wait for response* state. There, it checks two guards. First, it verifies if a reply arrived and if so, it processes the response. Otherwise, it checks if

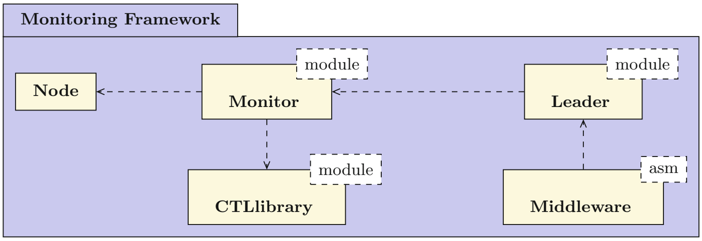


Fig. 5. Structure of the ASM monitoring specification

the request has a timeout. We had to let abstract concrete time details, but we replaced the timeout with a loop which can be executed a finite number of times (ten times in the case of our simulation). If the request has a timeout, it is stopped and the node is considered unavailable. The monitor moves, thus, to the *Report problem* state. After processing the request response, the monitor is ready to *Collect data* and after it finishes this process it moves to the *Retrieve information* state, where it tries to access additional data about the node. If the repository is not available, it carries out a diagnosis based on the current data, otherwise it queries the repository and executes a more complex analysis. If a problem has been discovered after analysis, the monitor moves to the *Report problem* state. Otherwise, it logs the data. The *Report problem* state corresponds to announcing the leader that it needs to a carry out a collaborative evaluation. After logging the data related to the current monitoring cycle, a guard verifies if its confidence degree is higher than the minimum accepted. In this case, the monitor starts a new cycle, otherwise it is deactivated by the middleware agent.

Table 1. Correlation between monitored data and node state

	Latency [ms]	CPU usage [percentage]	Memory usage [percentage]	Storage usage [percentage]	Work capacity [percentage]
Normal	<100	<40	<40	<40	>85
Critical	<100	>40	>40	>40	<85
Failed	>100	NA	NA	NA	NA

Data collected by the monitors capture a small set of parameters reflecting usage of resources and response time, which indicate possible unavailability and failure problems. We considered three possible diagnoses established by the monitors, $\{Normal, Critical, Failed\}$. Normal state corresponds to a small latency, small resource usage and high work capacity. Critical state refers to small latency, but high resource usage and small work capacity. Failure points to unavailability indicated by a high latency. The correspondence between the metrics collected by the monitor and the diagnosis it sets is depicted in Table 1. The values used for the classification are the ones expected generally for a service accessed through a high speed internet connection.

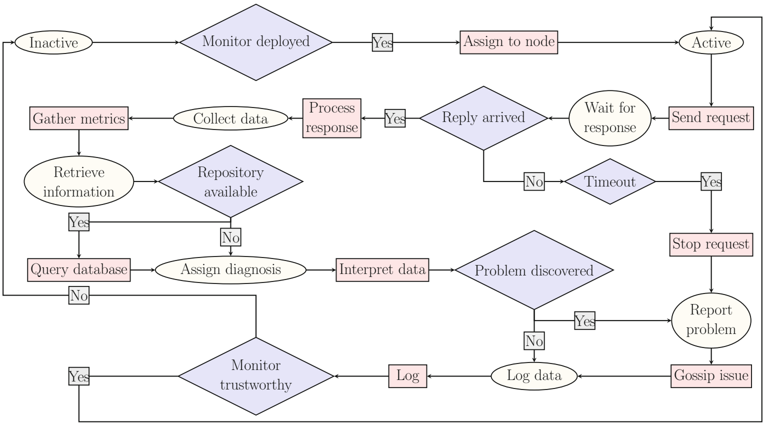


Fig. 6. Ground model of the monitor module

The rule responsible for collecting data from the node is captured in Code 1. It iterates through a set of specified metrics and for each of them checks if a value is defined and adds it to the gathered data.

```
rule r_GatherMetrics ($m in Monitor) =
  let ($count = 0) in
    while ($count < size(Metric)) do
      seq
        if (isDef(metric_value)) then
          monitor_measurements ($m) := append(monitor_measurements($m), (at
                (asSequence(Metric),iton($count)), metric_value))
        else
          monitor_measurements ($m) := append(monitor_measurements($m), (at
                (asSequence(Metric),iton($count)), 0))
        endif
        $count := $count + 1
      endseq
  endlet
```

Code 1. Monitor rule to gather metrics

If no value is defined, it simply adds zero, which is considered neutral by the specification. Data collection is done sequentially because a parallel execution tries to update the measurement list with different values at the same time. The inconsistency error was detected at simulation time with the aid of the AsmetaS tool. We leave as a future work the elaboration of transaction specific operations, which would permit submitting simultaneously multiple monitored values to the list of metrics.

The leader module focuses on collecting information from all the monitors of a node and using it for a final decision in case of reported issues. It starts from a state in which it waits for the *Evaluation requested* guard to become true. This guard is activated when a monitor reports a problem and moves the leader to the *Evaluate* state from where it requests data from all the monitors of the node. Then it moves to the assess state, where it uses the voting algorithm to evaluate the node. The voting method is a consensus problem and in comparison with previous algorithms, we require that each voter (monitor) contributes to a weight equal to its confidence degree to the final decision. The control state ASM of the leader is illustrated by Fig. 7.

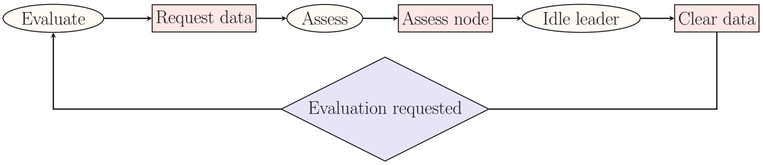


Fig. 7. Ground model of the leader module

The leader coordinates evaluations executed by different monitors assigned to the node. It, thus, aims to increase the accuracy of the monitoring services. We can analyze its role on the following example. Let us assume a node is observed by three monitors, all having the confidence degree equal to 100. One of them reports that the node is unavailable, making in this way the *Evaluation requested* guard. The real issue is that the communication link between it and the node is broken. In parallel, the other two monitors receive responses from the node and continue their assessment. On the request of the leader, they submit their evaluation and vote that there is no problem with the node. The leader analyzes all three responses and decides that the node does not exhibit any actual problem (by the voting procedure). The monitoring framework prevents in this way a false positive report of a problem, that would trigger an unnecessary adaptation of the node.

The middleware agent is expressed as an ASM where all the functions and instances are initialized. It contains only one state, *Executing*, and we assume that all the actions carried out by it are reliable and correctly executed.

5 Verification of the Model

5.1 Analysis of Model Quality

The AsmetaMA model reviewer tool establishes the quality of the model by checking its compliance to a set of properties and indicating which functions, rules and control states are not necessary or not well specified. The tool relies on the translation of the model to an NuSMV specification. We list below the list of properties we checked, but we refer the reader to [3] for the complete list of properties and more details on the review procedure.

1. MP1 - No inconsistent updates are performed.
   Result: NONE (NONE indicates that no violations have been found).
2. MP2 - Every conditional rule is complete
   Result: ConditionalRule if (heartbeat_timeout(self)) is not complete. ConditionalRule if (isUndef(has_leader($n))) is not complete.
3. MP4 - No assignment is always trivial
   Result: Trivial update of location trigger_gossip(MONITOR_3). When the condition is (TRUE & !(monitor_state(monitor_3) = INACTIVE) & (monitor_state(monitor_3) = LOG_DATA)) its value is always the same of the term FALSE.
4. MP5 - For every domain element e there exists a location which has value e
   Result: None
5. MP7 - Every controlled location is updated and every location is read.
   Result: Functions middleware_state, has_leader, assigned_node, confidence_degree could be defined static.

The result of the model quality analysis was used for refining the model and removing unnecessary functions or reviewing incomplete conditional structures.

5.2 Verification of Specification Properties

ASMETA toolset allows verifying properties of the model by using Computation Tree Logic (CTL) operators. The framework supports the translation of the model to an NuSMV specification that can be further checked. NuSMV supports only finite domains and simple data structures. Hence, the AsmetaL initial model had to be oversimplified. CTL properties were translated into AsmetaL functions, which are part of the *CTLlibrary* described by [2].

We carried out the verification of a set of properties, which handle aspects related to the communication between modules. We are interested in the correctness of the monitoring processes. We propose for the future work the inclusion of the confidence degree values in the verification process. As the verification phase is constrained to using finite sets, we defined one node to which we assigned three monitors having one leader. We simplified functions to contain finite Integer values instead of Real ones (for confidence degree or metric values) and focused on the correctness of the monitoring workflow. All the properties were specified in CTL and verified on a Windows machine having Intel(R) Core(TM) i7 CPU @ 2 GHz, 8 GB RAM with the aid of the AsmetaSMV Eclipse Plug-in.

Any monitor that is assigned to a node, being thus in the *Active* state, eventually reaches the state where he logs the information collected, *Log_data*. We ensure in this case that a monitoring cycle is eventually completed.

```
CTLSPEC (forall $m in Monitor with ag((monitor_state($m) = ACTIVE) implies
    ef(monitor_state($m) = LOG_DATA)))
```

If a monitor submits a request and does not receive a response for it, the monitor reports the issue. We ensure in this way, that unavailability of the node is communicated further.

```
CTLSPEC (forall $m in Monitor with ag((monitor_state($m) =
    WAIT_FOR_RESPONSE and not(heartbeat_response_arrived($m))) implies
    ax(monitor_state($m) = REPORT_PROBLEM)))
```

Monitors having a lower confidence degree are dismissed and move to the inactive state. This property does not allow faulty monitors to analyze nodes of the CELDS and aims to enforce a fail-safe behavior of the monitoring solution.

```
CTLSPEC (forall $m in Monitor with ag((confidence_degree($m) <
    min_confidence_degree ) implies ax(monitor_state($m) = INACTIVE)))
```

If any of the monitors reports a problem, the leader starts the evaluation of the node. The property verifies that all reported issues are handled by the system.

```
CTLSPEC (forall $m in Monitor with ag( (trigger_gossip($m) = true) implies
    ef(leader_state(leader_1) = EVALUATE )) )
```

A leader that starts the evaluation must reach a conclusion and establish an assessment. This property guarantees that the evaluation process provides a result to the system.

```
CTLSPEC (forall $l in Leader with ag((leader_state($l) = EVALUATE) implies
    ef(isDef(assessment($l)))))
```

```
> NuSMV -dynamic -coi -quiet C:\Work\Specs\ASMeta_Specs\code\Verification\SingleModelVerification.
    smv
-- specification ((AG (monitor_state(monitor_2) = ACTIVE -> EF monitor_state(monitor_2) = LOG_DATA) &
     AG (monitor_state(monitor_1) = ACTIVE -> EF monitor_state(monitor_1) = LOG_DATA)) & AG (
    monitor_state(monitor_3) = ACTIVE -> EF monitor_state(monitor_3) = LOG_DATA)) is true
-- specification ((AG ((!heartbeat_timeout(monitor_2) & (monitor_state(monitor_2) = WAIT_FOR_RESPONSE
    & heartbeat_response_arrived(monitor_2))) -> AX monitor_state(monitor_2) = COLLECT_DATA) & AG
    ((!heartbeat_timeout(monitor_1) & (heartbeat_response_arrived(monitor_1) & monitor_state(monitor_1) =
    WAIT_FOR_RESPONSE)) -> AX monitor_state(monitor_1) = COLLECT_DATA)) & AG ((!
    heartbeat_timeout(monitor_3) & (monitor_state(monitor_3) = WAIT_FOR_RESPONSE &
    heartbeat_response_arrived(monitor_3))) -> AX monitor_state(monitor_3) = COLLECT_DATA)) is true
-- specification ((AG ((monitor_state(monitor_2) = WAIT_FOR_RESPONSE & !heartbeat_response_arrived(
    monitor_2)) -> AX monitor_state(monitor_2) = REPORT_PROBLEM) & AG ((!
    heartbeat_response_arrived(monitor_1) & monitor_state(monitor_1) = WAIT_FOR_RESPONSE) -> AX
    monitor_state(monitor_1) = REPORT_PROBLEM)) & AG ((monitor_state(monitor_3) =
    WAIT_FOR_RESPONSE & !heartbeat_response_arrived(monitor_3)) -> AX monitor_state(monitor_3) =
    REPORT_PROBLEM)) is true
-- specification ((AG (confidence_degree(monitor_3) < 80 -> AX monitor_state(monitor_3) = INACTIVE) &
    AG (confidence_degree(monitor_1) < 80 -> AX monitor_state(monitor_1) = INACTIVE)) & AG (
    confidence_degree(monitor_2) < 80 -> AX monitor_state(monitor_2) = INACTIVE)) is true
-- specification ((AG (trigger_gossip(monitor_2) -> EF leader_state(leader_1) = EVALUATE) & AG (
    trigger_gossip(monitor_3) -> EF leader_state(leader_1) = EVALUATE)) & AG (trigger_gossip(monitor_1)
    -> EF leader_state(leader_1) = EVALUATE)) is true
-- specification AG (leader_state(leader_1) = EVALUATE -> EF assessment(leader_1) != undef) is true
-- specification AG (leader_state(leader_1) = IDLE_LEADER -> EF assessment(leader_1) != undef) is false
```

Code 2. AsmetaSMV Trace

A leader that is in idle state is not allowed to assign a diagnosis. This property checks that the leader does not misbehave and starts to randomly carry out assessments. The property must be evaluated to false and the tool offers a counterexample as well.

```
CTLSPEC (forall $l in Leader with ag((leader_state($l) = IDLE_LEADER)
    implies ef(isDef(assessment($l)))))
```

The AsmetaSMV result of the property verification is captured by the snippet from Code 2.

6 Discussions and Limitations

The current ASM specification focuses on achieving correct behavior of the monitors. It emphasizes the importance of establishing an accurate diagnosis of a component of the CELDS, given a set of partial views of the system submitted by each monitor assigned to the component. Although the ASM method permits a straightforward translation of the requirements into a formal model, our approach suffers of several limitations as follows.

In the validation stage, we had to simplify the specification by removing the non-deterministic character of the *choose* rules. The verification process implied translating the model to an NuSMV specification, which could be model checked. Hence, infinite domains had to be removed or replaced by finite sets of Integer/-Natural or enumerations. Another problem we encountered was the impossibility of assigning the value of a function as parameter for another function. Time related aspects of the solutions were also not supported by the ASM method. The ASM models can be complemented by other formal specifications focused on timing aspects like TLA+, for example.

The simplification of the model widens the gap to the high complexity of CELDS. However, the specification still captures important insights on the behavior of the monitors and helps identifying design flaws.

The research work described in this paper encompasses the steps executed to transform the requirements of the monitoring processes for CELDS to an actual formal model, which can be analyzed and verified. Once we had the ASM ground model, we could easily translate the states and rules to an AsmetaL model. The ASMETA toolset integrated with the Eclipse plug-in supported the simulation, validation and verification of the model, which were carried out gradually. However, there are still a number of open questions to be answered. For instance, the model adviser tool identifies unnecessary functions or incomplete conditional structures, but it does not assess the coverage of the model with respect to the problem domain. Validation by scenarios provides useful insights as long as the scenarios defined by the modeler are representative enough. In the verification phase, the understanding of the properties by the modeler determines the importance of model checking results. We consider that ASMETA toolset supports the designer in elaborating the specifications, assessing their quality, and eventually finding related drawbacks, but it still needs the human expertise in order to provide relevant answers.

7 Conclusions and Future Work

The paper addresses the monitoring aspect for CELDS from a formal perspective. We presented the methodology for elaborating and assessing a formal model for monitoring processes of CELDS using the ASMETA toolset. The focus of our work was on ensuring the correctness of the monitors, and hence enhancing the reliability and availability of the whole system. The work focuses on translating monitoring related processes to a formal model and verifying its properties.

Through rigorous analysis of the model we can identify design flaws, that otherwise, would propagate to the implementation phase of software development.

The model we propose is still open to future refinements, in which interpretation of data can be improved by considering aggregation of parameters into higher-level metrics. The spectrum of properties to verify can be enlarged by elaborating more the dependencies between the components and encompassing quantitative aspects of the voting process.

References

1. Arcaini, P., Riccobene, E., Scandurra, P.: Modeling and analyzing MAPE-K feedback loops for self-adaptation. In: 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-managing Systems, pp. 13–23, May 2015
2. Arcaini, P., Gargantini, A., Riccobene, E.: AsmetaSMV: a way to link high-level ASM models to low-level NuSMV specifications. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 61–74. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11811-1_6
3. Arcaini, P., Gargantini, A., Riccobene, E.: Automatic review of abstract state machines by meta property verification. In: Proceedings of Second NASA Formal Methods Symposium - NFM 2010, 13–15 April 2010, Washington D.C., USA, pp. 4–13 (2010)
4. Arcaini, P., Holom, R.-M., Riccobene, E.: ASM-based formal design of an adaptivity component for a cloud system. Form. Asp. Comput. **28**(4), 567–595 (2016)
5. Bianchi, A., Manelli, L., Pizzutilo, S.: An ASM-based model for grid job management. Informatica (Slovenia) **37**(3), 295–306 (2013)
6. Bolis, F., Gargantini, A., Guarnieri, M., Magri, E., Musto, L.: Model-driven testing for web applications using abstract state machines. In: Grossniklaus, M., Wimmer, M. (eds.) ICWE 2012. LNCS, vol. 7703, pp. 71–78. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35623-0_7
7. Börger, E., Stark, R.F.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer-Verlag, New York Inc., Secaucus (2003). https://doi.org/10.1007/978-3-642-18216-7
8. Börger, E.: Modeling distributed algorithms by abstract state machines compared to petri nets. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) ABZ 2016. LNCS, vol. 9675, pp. 3–34. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33600-8_1
9. Bósa, K., Holom, R.-M., Vleju, M.B.: A formal model of client-cloud interaction. In: Thalheim, B., Schewe, K.D., Prinz, A., Buchberger, B. (eds.) Correct Software in Web Applications and Web Services, pp. 83–144. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17112-8_4
10. Buga, A., Nemes, S.T.: A formal approach for failure detection in large-scale distributed systems using abstract state machines. In: Benslimane, D., Damiani, E., Grosky, W.I., Hameurlain, A., Sheth, A., Wagner, R.R. (eds.) DEXA 2017. LNCS, vol. 10438, pp. 505–513. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-64468-4_38
11. Buga, A., Nemes, S.T.: Towards modeling monitoring of smart traffic services in a large-scale distributed system. In: CLOSER 2017 - Proceedings of the 7th International Conference on Cloud Computing and Services Science, 24–26 April 2017, Porto, Portugal, pp. 455–462 (2017)

12. Buga, A., Nemes, S.T.: Towards modeling monitoring services for large-scale distributed systems with abstract state machines. In: Radar Track at the 22nd International Working Conference on Evaluation and Modeling Methods for Systems Analysis and Development (EMMSAD) Co-located with the 29th International Conference on Advanced Information Systems Engineering 2017 (CAiSE 2017), June 2017, Essen, Germany, vol. 1859. CEUR (2017)
13. Clayman, S., Galis, A., Mamatas, L.: Monitoring virtual networks with Lattice. In: Network Operations and Management Symposium Workshops (NOMS Wksps), pp. 239–246. IEEE/IFIP, April 2010
14. Fugini, M., Siadat, H.: SLA contract for cross-layer monitoring and adaptation. In: Rinderle-Ma, S., Sadiq, S., Leymann, F. (eds.) BPM 2009. LNBIP, vol. 43, pp. 412–423. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12186-9_39
15. Kossak, F., Mashkoor, A.: How to select the suitable formal method for an industrial application: a survey. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) ABZ 2016. LNCS, vol. 9675, pp. 213–228. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33600-8_13
16. Ma, H., Schewe, K.D., Wang, Q.: An abstract model for service provision, search and composition. In: 2009 IEEE Asia-Pacific Services Computing Conference (APSCC), pp. 95–102, December 2009
17. Moreno-Díaz, R., Pichler, F., Quesada-Arencibia, A. (eds.): Computer Aided Systems Theory - EUROCAST 2015. Lecture Notes in Computer Science, vol. 9520. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-27340-2
18. Nemes, S.T., Buga, A.: Towards a case-based reasoning approach to dynamic adaptation for large-scale distributed systems. In: Aha, D.W., Lieber, J. (eds.) ICCBR 2017. LNCS (LNAI), vol. 10339, pp. 257–271. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61030-6_18
19. Nemes, S.T., Buga, A.: Towards modeling adaptation services for large-scale distributed systems with abstract state machines. In: Proceedings of the Seventh International Symposium on Business Modeling and Software Design, BMSD, vol. 1, pp. 193–198. INSTICC, SciTePress (2017)
20. Németh, Z.N., Sunderam, V.: A formal framework for defining grid systems. In: 2002 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, p. 202 (2002). http://dblp.uni-trier.de/rec/bibtex/conf/ccgrid/NemethS02
21. Petcu, D., Crăciun, C., Neagul, M., Panica, S., Di Martino, B., Venticinque, S., Rak, M., Aversa, R.: Architecturing a sky computing platform. In: Cezon, M., Wolfsthal, Y. (eds.) ServiceWave 2010. LNCS, vol. 6569, pp. 1–13. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22760-8_1