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

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

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

Abstract: The evolution of Large-Scale Distributed Systems favored the development of solutions for smart cities. Such systems face a high-level of uncertainty as they consist of a large number of sensors, processing centers, and services deployed along a wide geographical area. Bringing together different resources poses increased complexity as well as communication efforts, and introduces a large set of possible failures and challenges of continuously growing computational and storage expectations. In such a frame, the role of the adaptation components is vital for ensuring availability, reliability, and robustness. This paper introduces a formal approach for modeling and verifying the properties and behavior of the adaptation framework addressing the case of a system failure. We formalize the behavior and the collaboration mechanisms between agents of the system with the aid of Abstract State Machines and employ the ASMETA toolset for simulating and analyzing properties of the model.

1 INTRODUCTION

Large-scale distributed systems (LDS) have appeared as a solution to the continuously expanding computing and storage demands. Services offered through such architectures bring an increased value to the end client, but there are still many open questions posed by issues like heterogeneity, network failures, and random behavior of components. Recovering from failures and ensuring a high availability of the system requires reliable monitoring and adaptation techniques.

One of the biggest beneficiaries of LDS are the applications for smart cities, which connect and allow the communication of a huge number of sensors spread along a wide radius. Such systems cover various aspects like traffic surveillance, infrastructure and environment, and aim to ease and improve the quality of life of inhabitants. These solutions are characterized by the same properties as well as failures and availability issues as any LDS. Therefore, the adaptation component plays a key role in enacting adaptation plans to bring the system to a normal execution mode.

The goal and contribution of this paper is to integrate the formal modeling capabilities of the Abstract State Machines (ASMs) for defining and validating an adaptation solution for LDS. Our project promotes a service-oriented approach to heterogeneous, distributed computing that enables on-the-fly run-time adaptation of the running system based on the replacement of sets of employed services by alternative solutions. For this we develop an advanced architecture and an execution model by envisioning and adapting a wide spectrum of adaptation means such as re-allocation, service replacement, change of process plan, etc.

The remainder of the paper is structured as follows. Section 2 provides an overview of the system and its architecture, followed by a description of the structure of the adaptation framework in Section 3. Essential concepts related to the Abstract State Machine formal methods as well as the formal specification of the adaptation framework are detailed in Section 4. Related work is discussed in Section 5, after which conclusions are drawn in Section 6.

2 SYSTEM OVERVIEW

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,

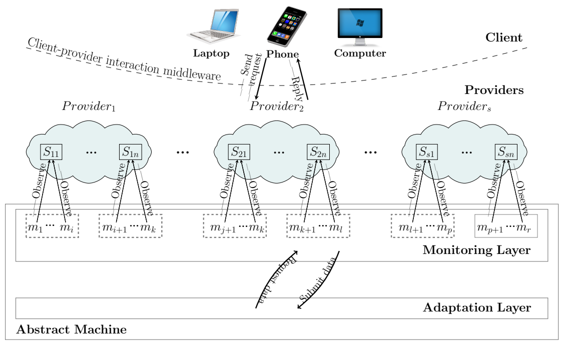

Figure 1: Architecture of the LDS system.

pollution, energy efficiency 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 traffic services, where centrally-controlled traffic sensors regulate the flow of traffic through the city in response to demand. The benefits of such a smart traffic management application also empowers people to take informed decisions, and prevent severe traffic congestion due to overcrowded areas. In a smart city network, traffic sensors provide real time data related to the percentage of road occupancy, the number or traffic participants, just to name a few. Such sensors are distributed along an LDS and the data they provide can be integrated with activity patterns extracted from smart gadgets for building a knowledge base for a traffic application.

The organization of the solution reflects the structure of LDS, where nodes refer to sensors and services are offered by various providers. Such systems are characterized by heterogeneity, unavailability, instability, network and node failures. Problems occurring at component level are propagated to the whole solution, making it hard to identify the source. We emphasize the role of the adaptation framework for ensuring availability of the system and propose a formal model for the solution. Figure 1 illustrates the architecture of the LDS system for a smart traffic application.

2.1 System Architecture

The envisioned solution proposes distributed middleware components containing different units responsible for specific tasks: service integration, process optimization, communication handler. The core makes use of ASMs for expressing the specification of the other components and foresees a three-layered abstract machine model addressing normal processing, monitoring and adaptation. The organization, as illustrated in Figure 1, is rooted in three parts: the client side where different users request services from providers, the side of the providers where sensors are deployed and an abstract machine containing the monitoring and adaptation layers for the resources of the providers. The interaction of the clients with the service providers is based on a solution defined by (Bósa et al., 2015), where the client-cloud interaction middleware processes the requests and ensures the delivery of services to the end user.

The processes of the monitors and adapters are highly interconnected and interdependent, enabling the system to perform reconfiguration plans whenever any of the traffic sensors faces a problem. The monitors are responsible for collecting data, aggregating it into meaningful information and communicating observations about abnormal executions to the adaptation framework. The latter deals with recovering from anomalous situations, logging them, and finding the best remedy to restore the LDS to normal running mode. Diagnosis is strongly correlated with the high-level interpretation of collected data.

The adoption of LDS demands a deep understanding of the underlying infrastructure, its running mechanisms and uncertainties, as services may become unavailable or change, or network problems may impact negatively on the reliability and performance of the distributed system (Grozev and Buyya, 2014). Delivery of reliable services requires a continuous evaluation of the system state and adaptation in case of abnormal execution. Therefore primarily, two aspects are considered: resilience and fault tolerance. With respect to resilience the project targets system architectures that guarantee that a LDS keep running and producing desired results, even if some services become unavailable, change or break down. With respect to fault tolerance the project targets assessment methods that permit the detection of failure situations and adaptive repair mechanisms. Therefore for LDS, adaptability is a valuable and an almost inevitable process.

3 ORGANIZATION OF THE ADAPTATION FRAMEWORK

As aforementioned in section 2, the architecture and execution model are enhanced to capture dynamic adaptation of a LDS to changing environmental circumstances. The Adaptation Engine aims to perpetually react to the input measurements and notifications from the monitoring component and maintain its resiliency to gracefully handle and adapt to new contexts varying from network traffic fluctuations to un-

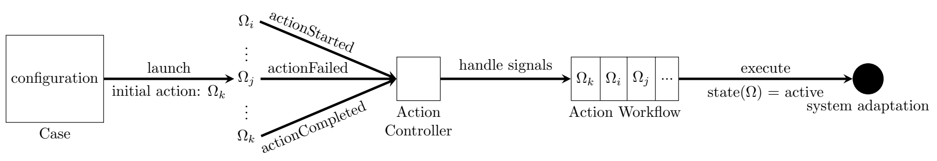

Figure 2: Overview specification of the Adaptation Manager.

availability of different system components. Its main measures consist in reacting to and evaluating the data collected and assessed by the monitoring components in regards to the detected faults within the system, employing the repair of the encountered problem under presumably optimal performance and adjusting the solution to higher levels of quality compliance.

Standardized repair actions for on-the-fly changes in reaction to identified critical situations are defined and will be employed on demand. This will result in a catalog of possible adaptive collaboration patterns, each supported by a set of subsequent adaptation tools and components. Such repair patterns can be the replacement of a component service by an equivalent one or the change of location for a service, up to the replacement of larger parts of the LDS, i.e. a set of services involved, by a completely different, alternative solution.

These steps in the adaptation process highlight the two major components that make up the Adaptation Engine as an inner component of the abstract machine included in the middleware: solution exploration, identification and maintenance carried out by the Case Manager, and solution management and enactment processed by the Action Manager. Each constituent component runs with well delimited responsibilities and areas of inference and control. The current paper focuses on the second part of adaptation, the Action Manager.

In the envisioned framework, any adaptation solution is configured and stored in the repository as a workflow schema detailing the actions and underlying transition dependencies needed to restore the system to a normal execution mode.

An action is an autonomous entity (e.g. a software module) which has the power to act or cause a single update to the system. Its autonomy implies that its processes are neither controlled by other actions, nor are they controlled by the environment. The power of such autonomous and self-aware actions lies in their ability to deal with unpredictable, dynamically changing, heterogeneous environments while relying fully on existing solutions for LDS adaptability. Given the situation of replacing one service with another, one such action would encompass finding a suitable matching service to replace the problematic one (by accessing the capabilities of an existing tool) or dynamically reconfigure the service calls to the new in use service.

The action's instantiation and execution are handled by linked ActionController loaded based on the defined contract for that particular action. The actions' ordering and dependency on other actions is handled by means of notification/signaling, where every action state change would imply for the parent ActionController to broadcast the associated notification. Figure 2 depicts the overall structure of the adaptation process once the problem is mapped to previous encountered problems and the attached solution is carried out based on its configuration.

Therefore, the adaptation system consists of a finite set of autonomous, interacting Action Controllers that intercept and assess all the raised notifications triggered by actions' execution or failure. The assessment implies either enacting and executing its corresponding action, or ignoring the notification as it is not of interest in the given solution configuration. Having Action Controllers to monitor and handle the interaction between the actions of a solution, it emphasizes new properties of the actions being defined in terms of needed input, concrete implementation and resulting output. More importantly, the underlining actions can be easily reused or substituted by enabling the possibility to add or remove any given number of actions without the need to update the current actions.

The model's underlining observer/controller architecture is one realization of the feedback loop principle (Brun et al., 2009): the executing adaptation is observed by its registered controllers, which in turn, based on the reported observations and broadcast notifications, affect the system towards the remediation of the reported problem/failure. An environmental change resulted from the execution of an adaptation action triggers a reaction within the system that causes, in return, a configuration-based chain of subsequent changes. These loops guide the system behavior and dynamics for the adaptation to succeed in reaching the intended goals.

In order to better understand the intrinsic problems that the framework can face, we focused our attention on building ground models in terms of ASMs. Based on them we can validate the specifications and verify if they fulfill desired properties as safety and liveness.

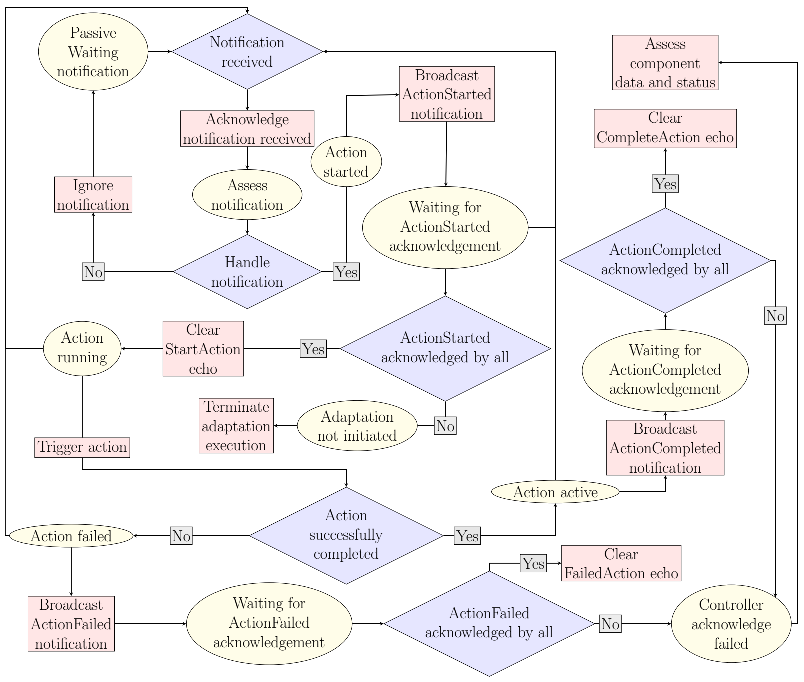
Passive Waiting notification
Notification received
Assess component data and status
Broadcast ActionStarted notification
Acknowledge notification received
Action started
Clear CompleteAction echo
Assess notification
Ignore notification
Yes
Waiting for ActionStarted acknowledgement
ActionCompleted acknowledged by all
No
Handle notification
Yes
No
Action running
Clear StartAction echo
Yes
ActionStarted acknowledged by all
Waiting for ActionCompleted acknowledgement
Terminate adaptation execution
Adaptation not initiated
No
Broadcast ActionCompleted notification
Trigger action
Action active
Action failed
No
Action successfully completed
Yes
Yes
Clear FailedAction echo
Broadcast ActionFailed notification
Waiting for ActionFailed acknowledgement
ActionFailed acknowledged by all
No
Controller acknowledge failed

Figure 3: Control ASM for the Action Controller agent.

4 FORMAL SPECIFICATION OF THE SYSTEM

4.1 Background on ASM Theory

According to (Kossak and Mashkoor, 2016), ASMs stood out as a high-quality software engineering method for behavioral and architectural system design and analysis. By further considering the assistance of the model through the expressiveness of the specification, the software development process, its coherence and the scalability in industrial applications (Kossak and Mashkoor, 2016) we adopted the ASM method.

The specification of an ASM consists of a finite set of *transition rules* of the type: **if** *Condition* **then** *Updates* (Börger and Stark, 2003), where an *Update* consists of a finite set of assignment $f(t_1, ..., t_n) := t$. 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 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.

Definition 1. *A* control state ASM *is an ASM built on the following rules : 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.*

In the design phase of software development, ASM technique permits transforming the requirements from natural language to ground models, and further to control state diagrams, that are easier to formalize. ASMETA [1] toolset for simulating, validating and model-checking ASM models, permits elaborating the models with the aid of the AsmetaL language, which is able to capture specific ASM control structures and functions. The models are further simulated and validated for inconsistencies. The tool permits also automatic review of the model for properties like conciseness or faultiness or for design issues. In the verification stage, properties like reachability, safety and liveness are defined and checked.

4.2 ASM Specification

Based on the overall specification of the adaptation framework mentioned in Section 3, we define the specific states and transitions of the adaptation processes, with emphasis on the management and enactment of

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

adaptation actions. The model contains ActionController ASM agents, each of which carrying out its own execution. The ground model illustrated in Figure 3 details the behavior of an ActionController in relation to the received and broadcast notifications. The ActionController can pass through several states by various rules and guards.

At initialization, the ActionController is in the *Passive, Waiting notification* state. This initial state is reached again either when the associated action's execution and acknowledgment by all the other controllers are fulfilled or when the received notification is not bound to influence the ActionController in question. This is a clear indication of the continuous character of the adaptation process which takes place in the background of service execution.

Once a notification arises, the ActionController acknowledges the received notification in disregard of the actual sender, after which it moves to the *Assess notification* state. The rule responsible for acknowledging a notification is captured in Listing 1.

```
rule r_AcknowledgeNotificationReceived($c in Controller,$broadcaster in Controller) =
  if (controller_state($c) = NOTIFICATION_RECEIVED) then
    seq
      controller_state($c) := ASSESS_NOTIFICATION
      par
        acknowledged_controllers($broadcaster) := acknowledged_controllers(
                $broadcaster) + 1
        r_HandleNotification[$c]
      endpar
    endseq
  endif
```

Listing 1: Acknowledge notification ASM rule.

Handling the received notification implies to broadcast first the notification that the action execution is bound to start, as captured in Listing 2.

```
rule r_BroadcastNotification($c in Controller, $n in Notification) =
  forall ($neighbor in Controller) then
    if (not(id($c) = id($neighbor))
      seq
        acknowledged_controllers($c) := 1
        par
          controller_state($c) := WAITING_FOR_ACKNOWLEDGEMENT
          AcknowledgeNotificationReceived[$neighbor, $c]
        endpar
      endseq
    endif
  endforall
```

Listing 2: Broadcast notification ASM rule.

The controller must act on executing the underlying action only once the notification is acknowledged by all neighboring ActionControllers which were instantiated as part of the same adaptation session. Depending on the output of the executed action, one notification will be broadcast signaling the success or failure of this particular system update. As there is no linked track of the ActionControllers' order to execution, if at least one ActionController does not acknowledge any of the sent notifications, the adaptation is abruptly terminated and the component data and status are assessed and logged accordingly. The rule responsible for triggering the associated adaptation action is captured partially in Listing 3.

```
rule r_TriggerAction($c in Controller) =
  seq
    while (controller_state($c) = RUNNING_ACTION)
      wait
    if (action_completed($c))
      par
        r_BroadcastNotification[$c, ACTION_COMPLETED]
        r_AwaitAcknowledgement[$c]
        if (acknowledged_controllers($n) = numberOfControllers)
          par
            r_ClearNotificationEcho[$c]
            controller_state($c) := WAITING_NOTIFICATION
          endpar
        else
          par
            controller_state($c) := CONTROLLER_ACKNOW_FAILED
                    AssessDataAndStatus
          endpar
        endif
      endpar
    else
      par
        BroadcastNotification[$c, ACTION_FAILED]
        ...
```

Listing 3: Trigger action ASM rule.

4.3 Validation of the Model

The validation for the current state of our work deals only with the separate processes for each agent. It focuses on checking the work ow and the transitions from different states. AsmetaV tool permits validation of specific scenarios defined with the aid of the Avalla language presented by (Carioni et al., 2008). Scenarios resemble the unit tests performed during the software testing development phase. They capture execution flows given specific values to functions of the system.

One of the problems we identified during the validation phase was that the Avalla language does not support working with infinite domains. Therefore, we needed to consider that each ActionController remembers only one Notification instance. Other inconsistency errors detected at simulation time led to design changes or restrictions.

More than one system failure can be reported in a short time frame. Therefore, the failure recovery part is done in a sequential mode because, although the case/reconfiguration plan is locked while it's associated solution is executed, a parallel execution of simultaneous adaptations may try to update system parts or components with different values at the same time. We leave as a future work the elaboration of transaction specific operations, which would permit triggering simultaneously multiple adaptions within the system. This could be supported by annotating the case with extensive knowledge on the area of inference in the system of each case, which would later on be considered in the retrieval phase of the process.

5 RELATED WORK

Research in software adaptation ranges from the development of generic architectural frameworks to specific middleware using component frameworks and reflective technologies for specialized domains. Mechanisms proposed include: DA by generic interceptors (Sadjani, 2004), which do not modify a component's behavior, but intercept messages between components; DA with aspect-orientation (Yang, 2002); parametric adaptation (Pellegrini., 2003) or dynamic reconfiguration by means of adjusting or fine-tuning predefined parameters in software entities; dynamic linking of components (Escoffier and Hall, 2007); and model-driven development (Zhang and Cheng, 2006).

However, while existing techniques offer a wide range of options to achieve different degrees of DA, questions related to the identification and soundness of a given adaptation model are still open. Formal methods grant clearer definitions and precision for the adaptation framework. Our project focuses on how to extend and build on this previous research while specifying and validating LDS specific requirements like on-the-fly reaction to change, loss or addition of resources. We consulted the area of formal methods and chose the ASM technique proposed and exemplified in various industrial examples by (Börger and Stark, 2003).

Modeling LDS has been addressed in several cloud and grid related projects. The ASM technique contributed to the description of the job management and service execution in (Bianchi et al., 2013). Specification of grids in terms of ASMs have been proposed also by (Németh and Sunderam, 2002), where the authors focused on expressing differences between grid and traditional distributed systems.

6 CONCLUSIONS

The current paper proposes an approach for achieving a reliable adaptation solution for LDS. By employing the ASM formal method we analyze the properties of the model and identify reasoning flaws. The knowledge scheme presented in the paper supports adaptation related processes and is reflected in the model. We analyzed the model with the aid of the AsmetaV tool and validated the reliability of some of our models when executing an adaptation solution.

In the future steps of our work we aim to enhance the models and express their properties in terms of CTL logic, which is supported by the Asmeta toolset. By these means, faults and drawbacks of the proposal can be identified and corrected.

REFERENCES

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.

Bósa, K., Holom, R., and Vleju, M. B. (2015). A formal model of client-cloud interaction. In *Correct Software in Web Applications and Web Services*, pages 83–144.

Brun, Y., Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Müller, H., Pezzè, M., and Shaw, M. (2009). Software engineering for self-adaptive systems. chapter Engineering Self-Adaptive Systems Through Feedback Loops, pages 48–70. Springer-Verlag, Berlin, Heidelberg.

Carioni, A., Gargantini, A., Riccobene, E., and Scandurra, P. (2008). 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. Springer-Verlag.

Escoffier, C. and Hall, R. S. (2007). *Dynamically Adaptable Applications with iPOJO Service Components*, pages 113–128. Springer Berlin Heidelberg, Berlin, Heidelberg.

Grozev, N. and Buyya, R. (2014). Inter-cloud architectures and application brokering: taxonomy and survey. *Softw., Pract. Exper.*, 44(3):369–390.

Kossak, F. and Mashkoor, A. (2016). *How to Select the Suitable Formal Method foran Industrial Application: A Survey*, pages 213–228. Springer International Publishing, Cham.

Németh, Z. N. and Sunderam, V. (2002). A Formal Framework for Defining Grid Systems. *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 0:202.

Pellegrini., M.-C., R. M. (2003). Component management in a dynamic architecture. *The Journal of Supercomputing*, 24(2):151–159.

Sadjani, S., M. P. (2004). An adaptive corba template to support unanticipated adaption. In *International Conference on Distributed Computing Systems*, pages 74–83.

Yang, Z., C. B. S. R. S. J. S. S. M. P. (2002). An aspect-oriented approach to dynamic adaptation. *WOSS*, pages 85–92.

Zhang, J. and Cheng, B. H. C. (2006). Model-based development of dynamically adaptive software. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 371–380, New York, NY, USA. ACM.