https://doi.org/10.31449/inf.v47i6.4622 Informatica **47** (2023) 75–96 **75**

ASM-based Formal Model for Analysing Cloud Auto-Scaling Mechanisms

Ebenezer Komla Gavua[1], Gabor Kecskemeti[2]
[1]Institute of Information Technology, Miskolc-Egyetemvaros, 3515, Miskolc, Hungary
[1]Department of Computer Science, Koforidua Technical University, Koforidua, Ghana
[2]Department of Computer Science, Liverpool John Moores University, Liverpool, UK
E-mail: gavua@iit.uni-miskolc.hu, g.kecskemeti@ljmu.ac.uk

Keywords: auto-scaling, abstract state machines, cloud computing algorithms; formal modelling

Received: January 18, 2023

The provision of resources to meet workloads demands has become a crucial responsibility for auto-scaling mechanisms (auto-scalers) on cloud infrastructures. However, implementing auto-scaling mechanisms on cloud frameworks has presented numerous technical challenges. A typical challenge is that, these auto-scalers are often designed on different cloud systems making their evaluation, comparisons and wider applicability problematic. To address this issue, we propose an Abstract State Machine (ASM) model as a solution. Our ASM model was developed systematically according to the behaviours of several auto-scalers, covering the necessary system processes. Rigorous validation and evaluation of our model have been conducted using the CoreASM Model Checker. The results demonstrate that our model can effectively analyze and generate accurate ASM refinements for auto-scalers, even without the need for real-life experiments. Our model, therefore, provides the platform to evaluate the behaviours of algorithms executed on clouds.

Povzetek: Razvit je model ASM, ki analizira in generira izboljšave ASM za avtomatske skalirnike.

1 Introduction

The dynamic provisioning of computational resources has emerged as a critical objective for many cloud applications designers. To achieve this, auto-scaling mechanisms (auto-scalers) are employed to ensure that resources effectively meet workload demands while upholding the reliability of virtual infrastructures [1, 2]. Research has identified many challenges with auto-scalers [3, 5]. These challenges predominantly stem from limited understanding of the behaviors exhibited by auto-scalers, particularly when they are developed across different frameworks.

In this paper, we advocate the use of Abstract State Machines (ASMs [8]) as a mathematically well-founded framework for analyzing and comparing auto-scalers designed on different cloud systems. ASMs, initially introduced by Gurevich as evolving algebras [4], have proven successful in cloud systems for formally designing adaptivity components [22].

The main contribution of this work is to introduce new ways of analysing and comparing auto-scalers designed on different clouds. The modelling process of our ASM model is presented in five steps. These are (i) design and analysis of the framework of our model. (ii) design and implementation of five ASM transition rules to reflect typical job execution phases. (iii) comparisons of auto-scalers offered alongside the DISSECT-CF cloud simulator [10], with our transition rules. (iv) validation of our model with test cases created from existing auto-scalers. (v) evaluation of our model with our transition rules, ASM refinement method, and evaluation goals. To accomplish these steps, first, we created the blueprint of our model while detailing state transitions during job processing. This blueprint comprises of a framework that represents categories of auto-scalers. Second, we described the details of our ASM rules and how they were implemented to reflect our model. Third, we examined algorithms developed from available auto-scalers to identify similar behaviours among them. Fourth, we validated test cases developed from available auto-scalers. The validation processes were accomplished with goals created from our model's transition rules. Our model was checked and validated with test cases using the CoreASM Model Checking Tool. Fifth, we evaluated our model with criteria developed from ASM definitions and methods to highlight the efficiency of our model.

The results of our validation and evaluation demonstrate that our model provides valuable insights into the behavior of auto-scalers, even without the need for real-life or simulated experiments. It offers a robust approach to analyze and compare auto-scalers designed for different cloud systems.

The remaining sections of this paper are structured as follows: In Section 2, we review relevant research on auto-scaling techniques and the application of ASMs in cloud and distributed systems. Section 3 presents our methodology for analysing these techniques. We then validate and evaluate the results of our modeling through ASM simulations in Section 4. Finally, in Section 5, we conclude the

paper and provide recommendations for future work.

2 Related works and background

This section discusses past research efforts to provision resources during auto-scaling, and the application of ASM on distributed and cloud systems.

2.1 Overview of auto-scaling research

The quest for auto-scaling of computing resources on clouds, has been due to deviations in expected resources versus actual resource usage [2, 13]. This deviation was analysed by [3], and in their work, auto-scaling techniques were classified into demand-oriented categories. These works propelled research activities into the provision of cloud resources to meet workload demands.

Ghanbari et al. [11] employed a stochastic, Model Predictive Control problem (MPC) technique to formulate an auto-scaling approach that exploits the trade-off between performance-related objectives and cost minimization. Yang et al. [7] investigated the problem of cost-aware auto-scaling along with predicted workloads in service clouds. They proposed an approach to scale service clouds in both real-time scaling and pre-scaling modes. Gandhi et al. [12] proposed a new cloud service, Dependable Compute Cloud (DC2), that automatically scales user applications in a cost-effective manner to provide performance guarantees. DC2 determines scaling actions for an application deployed in the cloud. Saxena et al. [14] proposed an integrated proactive auto-scaling and allocation of VMs approach. The solution allows load consolidation on few energy-efficient physical machines without affecting user application performance. Al-Dulaimy et al. [15] proposed a Service Level Agreement provisioning approach (MULTISCALER) that aims at the control of the contention and scaling of resources amongst co-hosted VMs. MULTISCALER handles changes in workloads auto-scaling in the presence of noisy neighbours. Ullah et al. [16] proposed a Cartesian genetic programming (CPG) based neural network (ANN) for resource utilisation estimation. The proposed system utilises a rule-based scaling system for elastic scaling of cloud resources.

Most of the auto-scaling research discussed above used statistical and experimental procedures. Less attention have been given to devising a flexible and formal approach [17] (such as ASMs) for evaluating auto-scalers. Let us now discuss the application of ASMs in clouds and other distributed systems.

2.2 Prior art with ASMs for clouds and other distributed systems

A few auto-scaling works have been undertaken relating to ASM modelling. LakshmiPriya *et al.* [19] proposed a formal framework based on ASMs, for specifying and defining an autonomous network layer grid. The model serves as a guideline to identify the minimum functionalities required of a grid. Bianchi *et al.* [20] proposed an ASM model to represent the standard mechanisms defined in Open Grid Service Architecture (OGSA) for job management and execution capability. The approach describes the components of a grid system, dynamical properties and relations between them in a service-oriented view. Bianchi *et al.* [21] proposed a formal framework for job execution management in grid systems and implemented it on the coreASM tool. The framework expresses the composition of interoperable, always refineable building blocks, which is an effective choice for defining a precise semantic foundation of a grid system. Arcaini *et al.* [22] proposed a ASM solution to tackle the problem of making cloud services usable to different end-devices. The framework consists of a server that intercepts requests from the clients and forwards them to the cloud.

The above discussion demonstrates limited research relating to formalizing auto-scalers. Thus we set out to devise a formal ASM model, which allows comparison of auto-scalers. But first, let us carry out a comparative analysis of the above related works with our proposed solution.

2.3 Comparative discussion of overview of related works

This section presents the comparative analyses of the research activities discussed in sub-sections 2.1 and 2.2 with our proposed approach. Table 1 discusses the summary of the results of the overview of the auto-scaling approaches.

Ghanbari et al. [11] formulated an approach that exploits the trade-off between performance-related objectives and cost minimization. However, the technique experiences challenges in resource provision due to delays associated with boot-up, running the initial installation scripts, and the initial warm-up. Also, the algorithms utilised have not been made public and not sufficiently analyse to foster comprehension and improvement. Therefore, an approach which allows in-depth analyses and comparisons of scaling algorithms will foster the efficient evaluation of auto-scaling approaches. Yang et al. [7] investigated the problem of cost-aware auto-scaling along with predicted workloads in service clouds. However, the approach is only applicable to service clouds. This inherent challenge limits the strategy's extension to several cloud platforms. Gandhi et al. [12] proposed DC2 that scales the infrastructure to meet the user-specified performance requirements. However, the method applied does not offer optimal estimates of the state of processes. Therefore, an approach that models the state transition of processes will allow users to evaluate state transitions during job processing.

Saxena et al. [14] developed an integrated proactive resource provisioning and allocation approach. However, the approach requires further work on tasks prediction and scheduling of VMs to reduce network traffic. Therefore, an approach that allows the modelling of the auto-scaling

Auto-Scaling Approaches	Results	Design Deficiency / Upgrade	Suitable for Analysis and Adaptability
Cost Minization [11]	This approach models application dynamics in clouds via the MPC technique.	This approach experiences delays linked with boot-up, running the initial installation and warm-up.	The shortfalls identified in this approach with resource provision does not foster extension to other platforms.
Workloads Prediction [7]	This approach predicts workload to scale virtual resources at different resource levels in service clouds.	Key state transitions during resource provision are omitted in the algorithms.	The design is limited to only service clouds and not extensible to other cloud platforms.
DC2 Approach [12]	DC2 scales the infrastructure to meet the user-specified performance requirements.	The service time estimated at runtime becomes problematic with some workloads.	This approach lacks optimal estimates of process states which prevents in-depth analysis.
Proactive Auto-Scaler [14]	This approach consolidates load on few PMs without affecting user application performance.	Tasks are scheduled on VMs which are distant. This causes excess network traffic, resulting in high energy usage.	The VMs locations affects resources provisions evaluation. Also, the algorithms are analysed into details to foster extension.
MULTISCALER Approach [15]	MULTISCALER allocates resources to VMs based on the SLA requirements.	Only vertical scaling is applied to meet applications performance goals by assigning the resources required.	The skewed nature of this approach affects its applicability to several situations.
Resource Estimation [16]	This approach estimates resources with recurrent CGP and ANN.	This strategy is designed with offline evolvability and coarse-grained scaling making it inaccessible.	The scaling algorithms are not analysed and integrated to workloads analysis, which is inimical to extension.

Table 1: Summary results of overview of related works.

of resources and the scheduling of VMs will improve the current design. Al-Dulaimy et al. [15] developed a novel Multi-Loop Control approach to allocate resources to VMs based on Service Level Agreements (SLA). However, their approach is limited to the provision of platform metrics such as CPU utilisation as input for scaling. Therefore, an approach that promotes the modelling of hybrid scaling will help service providers to design their platforms to meet the demands of a larger section of users. Ullah et al. [16] proposed a Cartesian genetic programming based neural network for resource utilisation estimation. However, the method utilized is not integrated with predictive scaling mechanisms for the analysis of workloads. Therefore, an approach that is capable of analysing auto-scalers with emphasis on VMs provisions, will enable authors to evaluate the shortfalls in their approach, for the upgrade of their current design.

Furthermore, table 2 discuss the summary of results of the state-of-the-art applications of ASM to clouds and other distributed systems discussed in sub-section 2.2.

LakshmiPriya *et al.* [19] developed a formal framework for an autonomous Network-Infrastructure for grids. However, the authors did not include validation techniques and refinement schemes for grids. Therefore, a model with flexible transition rules that includes formal framework with validation and refinement schemes will foster the application the modelling and validation processes to grids.

Bianchi *et al.* [20] utilized ASM modelling to study Grid systems as a composition of interoperable building blocks. However, the architectural specifications provided did not capture user requirements for performance monitoring. Therefore, a model that provides ASM refinements on user service requests and response for vertical and horizontal scaling will be essentially required for model adaptability and enhancements.

Bianchi *et al.* [21] also developed an ASM-based model for grid job management. However, the resource dispatching policy of the model requires further works. Therefore, an model that focuses on resources provisions behaviours and allows the comparisons of several situations will enable researchers and practitioners to evaluate all aspects of distributing computing and to provide the upgrades required.

Arcaini *et al.* [22] employed ASM to formally analyse a Client-Server (CS) adaptivity component for clouds. However, their design was limited to communications between client-server applications. Extensions were not provided to auto-scaling mechanisms. Therefore, a model that focuses on the auto-scaling of resources during job processing will allow users to examine their work for improvement.

ASMs on Clouds & Distributed Systems	Results	Design Deficiency / Upgrade	Suitable for Adapting to Other Frameworks
Autonomous Network-Infrastructure [19]	A formal framework for the minimum functionalities requirements for grids was proposed.	Model validation techniques and refinements schemes are unavailable.	This framework is not suitable for modelling resource provision behaviours in grids and clouds.
Grid Systems [20]	ASM was utilised to model the standard mechanisms defined in OGSA for job management and execution capability.	Complete architectural specification with requirements for grids to foster adaptability is unavailable.	The incomplete specifications in this approach makes the framework unadaptable to architectures.
Grid Job Management. [21]	A formal framework for job execution management in grid systems was developed.	The resource allocation policy which monitors resource provisions is incomplete.	The deficiencies in this approach makes it unadaptable to the frameworks of other distributed systems.
Client-Server Adaptivity Component. [22]	A formal framework was developed for cloud service provisioning to different devices with different profiles.	The design was limited to CS applications interactions. The Auto-Scaling of Resources on clouds was not considered.	The transition rules are not formalized to foster the flexible adaptation of the framework to cloud auto-scalers.
Proposed Model	A formal framework capable of analysing the virtual machine provision behaviours of auto-scaling mechanisms.	Model validation techniques and refinements schemes are available. Complete architectural specification with requirements for clouds.	This approach allows adoption of auto-scalers with extra features besides vertical and horizontal scaling to foster their evaluation.

Table 2: Summary results of overview of related works continuation

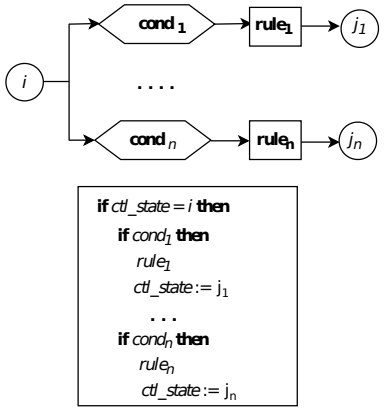

Figure 1: Control state ASMs

2.4 Abstract state machine theory

This sub-section reviews the theoretical background for ASMs. The ASM theory encompasses a formal system engineering technique that guides software development.

2.4.1 Abstract state machines

Abstract State Machines (ASM) are systems based on the concept of state transitions. They represent the rapid configurations of a system under development with transition rules. ASM *transition rules* express how function interpretations are modified from one state to another to reflect system changes. The basic form of a transition rule is the *guarded update*: "**if** *condition* **then** *updates*", where *updates* is a set of function of the form $f(t_1, ..., t_n) := t$. These updates are executed when a *condition* is true.

Definition 1. *A Control State ASM is a given control state i, where only one of the conditions $cond_k$ can be true for all $1 \leq k \leq n$. If the machine executes $rule_k$ and $cond_k$ to true. It changes the control state from i to j_k. These ASMs are mostly applied in the ASM refinement method.*

2.4.2 The ASM refinement method

The ASM refinement method is a stepwise refinement for crossing levels of abstraction. This method links models through well-documented incremental development steps. The steps start from ground models and turn them piecemeal into executable codes [23, 24]. Now, let us define a theorem for checking equivalence in our model.

Definition 2. Börger's refinement*: Given a notion $\equiv$ of equivalence, an ASM M^* is a correct refinement of an ASM M if and only if, for each M^*-run $S_0{}^*$, $S_1{}^*$,..., there is an M-run S_0, S_1, ...and sequences $i_0 \leq i_1 \leq \ldots$ and $j_0 \leq j_1 \leq \ldots$ such that $i_0 = j_0 = 0$ and $S_{ik} \equiv S_{jk}$ for each k and either, as seen in figure 3.*

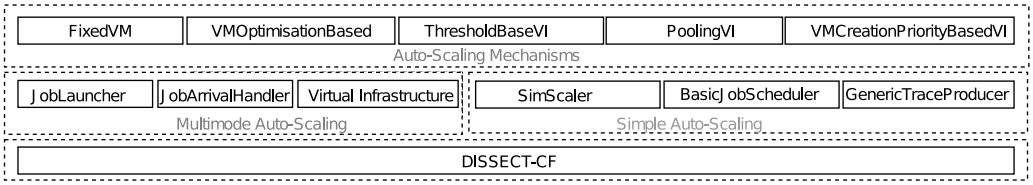

Figure 2: Architectural view of Auto-Scaling Mechanisms (ASMs) on DISSECT-CF.

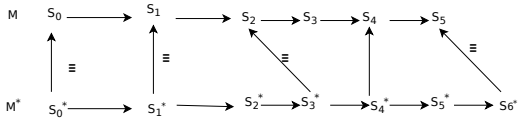

Figure 3: Borger's refinement

2.4.3 Ground model, universes and signatures

A *ground model* is a rigorous high-level system blueprint specification using domain-specific terms. All stakeholders can understand this model as it reflects the initial requirements and removes ambiguities of the initial textual conditions. The ground model is designed with *universes* and *signatures*. ASM depicts *universes* as basic sets with functions and relations. These sets require *signatures* for modelling state transitions. A signature is a finite set of function names, each of fixed arity. Now let's discuss our model's design in the next section.

3 Methodology

This section discusses our ASM model developed according to the following goals. First, we review the auto-scalers offered alongside DISSECT-CF. Second, we discuss our ground model, universe and signatures developed following the ground model discussion in sub-section 2.4.3. Third, we discuss our model's ASM functions developed per definition 1. Fourth, we discuss our model's development process per method 2.4.2 while comparing the categories of auto-scalers.

3.1 Auto-scalers on DISSECT-CF

This sub-section discusses the auto-scalers whose behaviours were utilized to establish our model. Many auto-scalers are evaluated through simulations. So, we have investigated one such simulation environment based on the DISSECT-CF simulator. We chose it as it has been shown through research to be efficient for auto-scaling experiments. In our model building, we have examined the simulator's auto-scaling related examples[1].

The chosen examples were built on several components. The auto-scalers observed can be grouped into two categories (i.e. *simple* and *multimode* auto-scalers). The *simple* auto-scalers respond to demands by either increasing or decreasing the VM instance counts. The scaling adjustments ensure that the quantity of VMs instances meets workload demands. The *multimode* auto-scalers also exhibit *simple* auto-scaler features. Furthermore, the *multimode* auto-scalers monitor the VM counts during scaling operations, which also influences the schedule resulting in the utilization of VMs. All auto-scalers are founded on a handful of classes including the Virtual Infrastructure (VI), JobArrivalHandler, BasicJobScheduler (BJS), GenericTraceProducer (GTP) and Simscaler.

VIs have the dedicated role of managing VMs belonging to particular applications. JobArrivalHandlers abstract the application model with the help of replaying customizable parts of pre-recorded workload traces. BasicJobScheduler combines with the job arrival handler functionality to process jobs. JobLaunchers and JobToVmSchedulers focus on the model for cluster scheduling techniques over the VMs offered by a particular VI.

Now let's discuss the auto-scalers offered with the simulator:

ThresholdBasedVI Mechanism is governed by a lower and an upper threshold. It observes VM utilisation and makes decisions based on how it relates to the two thresholds. It removes VMs not used to some extent to the lower threshold. In contrast, it adds new VMs when most of the VMs in the managed infrastructure are utilised more than the upper threshold.

VMCreationPriorityBasedVI Mechanism is a variation of the above approach, by anticipating growth in the infrastructure utilisation. It focuses on creating VMs mainly and only removes VMs as a last resort.

PoolingVI Mechanism is designed to keep a given number of completely unused VMs for newly arriving jobs. Hence, it can accept a new job anytime.

VMOptimisationBased Mechanism allows VMs created for one kind of executable to be repurposed to execute others, fostering VM reuse.

FixedVM Mechanism is designed to provision VMs for scaling jobs at minimal system resources. It utilizes a scaling mechanism (simscaler) on a production cloud infrastructure. The simscaler combines with the GenericTraceProducer to provision VMs for jobs.

[1]available at `https://github.com/kecskemeti/dissect-cf-examples` and at `https://github.com/kecskemeti/dcf-exercises`

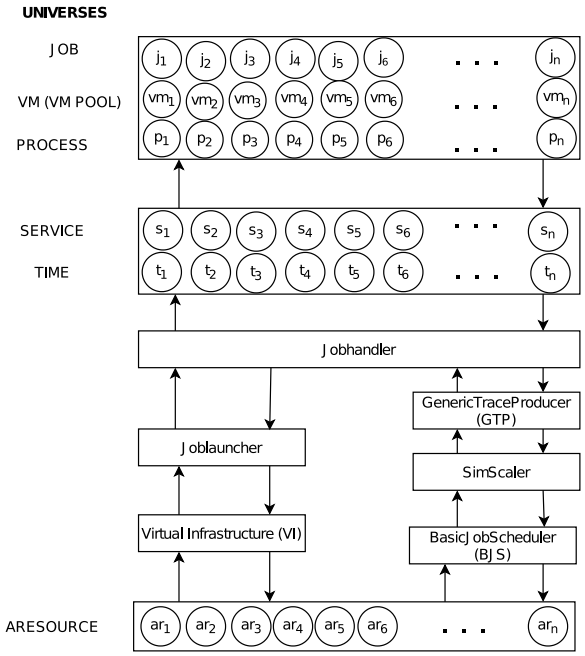

Figure 4: Basic elements of the ASM model for Auto-Scaling

For ease of discussion, the above mechanisms will be referred to as Threshold, Vmopt, Vmcreate, Pooling and FixedVM.

3.2 Model design

This sub-section discusses the modelling processes of our approach. This is presented in five design stages:

Design and Analysis of the model's framework as displayed in figures 4 and 5. The framework shows our model's ground model for the two categories of auto-scalers. Figure 4 shows the basic elements (*universes* interacting with *signatures* with arrows) utilised in designing our model. The arrows represent the relations between the *signatures* and the *universes* while provisioning resources during multimode or simple auto-scaling. The *signatures* are declared through function created in accordance with sub-section 2.4.3. Figure 5 shows *universes* interacting with unidirectional and bidirectional arrows. The bidirectional arrows represents information flow between universes while the unidirectional arrow represent the expected state changes during ASM runs.

Design and implement the model's *ASM Transition Rules* to reflect the job execution phases. Five ASM rules were defined and discussed in conjunction with the ASM method 2.4.2 and definitions 1 and 2.

Compare algorithms from the two categories of auto-scalers offered with DISSECT-CF, with *Transition Rules*. This step was modelled simultaneously with the previous step to ensure model coherence.

Model Validation with validation goals on test cases created from existing auto-scalers. The test cases are abstracted from the formalized algorithms of our auto-scalers to determine if the algorithms satisfy the requirements of our ground model.

Evaluation of the model with the *Transition Rules* and evaluation goals. This process was achieved in conjunction with the ASM refinement method (an ASM benchmark). The evaluation goals were employed to foster the applicability of our model to existing auto-scalers and to foster their equivalence to our ground model. The evaluation goals were selected in accordance to our transition rules and the control state ASMs.

Now let's describe our identified universes:

The *JOBHANDLER* is the universe that processes traces and sends its jobs to a joblauncher in the multimode auto-scaling mechanisms. In the simple auto-scalers, it is in charge of sending jobs to VMs for processing.

The *JOBLAUNCHER* is the universe that emits jobs for processing for the multimode auto-scalers. This deals with the ordering and timing of new jobs before they are released for processing.

A *JOB* The data submitted by a Jobhandler to be executed on a node. It contains binaries of an application, libraries and resource descriptions.

The *ARESOURCE* represent the major resources required for job processing. These include virtual infrastructures, cloud service, and hardware disk requirements. Different categories of resources (i.e. heterogeneous nodes) are also part of *Aresources*.

The *TIME* The duration a submitted job must spend being processed by virtual machines. This is usually measured in seconds.

The *VM* The virtual machine required for processing jobs in an auto-scaled infrastructure.

The *VI* This universe is responsible for managing VMs for applications in multimode environments.

The *PROCESS* This universe is responsible for ensuring that installed tasks receive the necessary attention from the *Aresources* and the VM.

The *SERVICE* This universe is responsible for service provision in multimode environments. Services are supplied per user requests at specific times on service clouds.

The *SIMSCALER* This universe is responsible for ensuring basic scaling activities in simple auto-scaled environments.

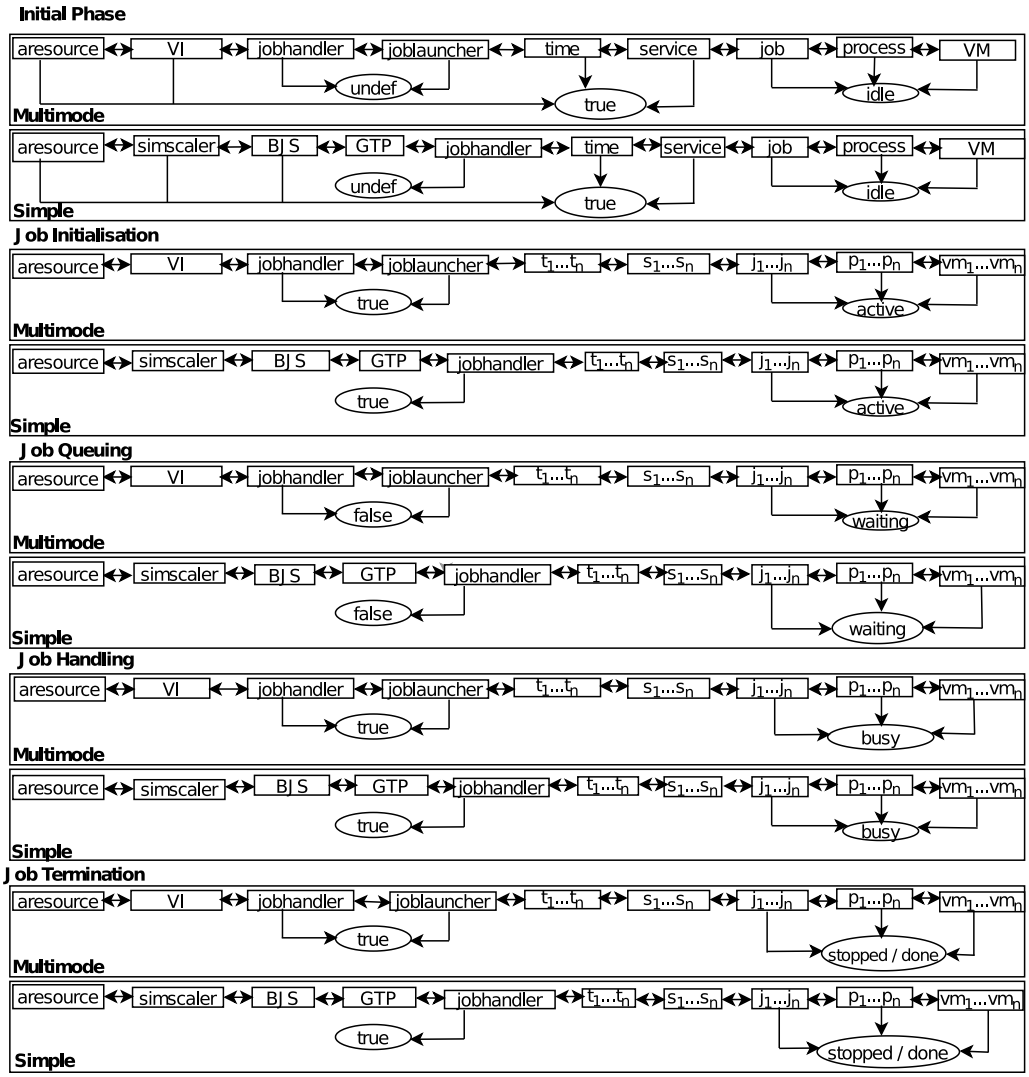

Figure 5: Ground model for ASM auto-scaling

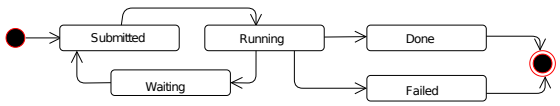

Figure 6: Job state transitions

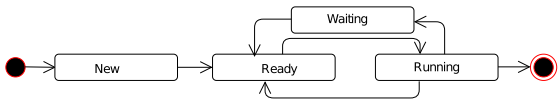

Figure 7: Process state transitions

The *BASICJOBSCHEDULER* utilises clustering patterns to monitor VMs for simple auto-scalers.

The *GENERICTRACEPRODUCER* provisions sets of jobs for processing for specific durations in simple auto-scaled environments.

3.3 ASM functions

Our ASM functions and corresponding state diagrams are depicted in tables 3 and figures 7 and 6 respectively.

JobState depicts the state transitions of job during data processing. Jobs' states transition from submitted to either done or failed as shown in figure 6.

JobTime depicts periods reflective of job states during task processing. It combines with *JobState* to describe at what period a particular state change occurred.

ProcessState illustrates process state transitions from new to stopped during job processing as shown in figure 7.

JobRequest invokes jobs generation which are mapped to VMs for task processing. It combines with *process-Request* to maintain a job and process requests during job initialising and handling.

MappedVM monitors the state of VMs and jobs connection for job processing. It combines with *MappedJob* to maintain the link between jobs and VMs.

Belongsto ensures that there is enough aresources to support a VM before it is selected. *Compatible* ensures that the VM selected is the appropriate one. This is done to prevent the selection of VMs with less utilisation which can be destroy within a short period.

AddVM combines with *Compatible*, and *BelongsTo* to foster VMs selection during job queuing.

DestroyVM is activated by auto-scalers to monitor VM utilisation and to remove unused VMs. Additionaly, it is used by certain auto-scalers to monitor the duration VM utilisation.

SystemRequest is the refinement for ProcessRequest and JobRequest. *ReqResources* is the refinement for MappedVM and MappedJob.

JobProcessing is the refinement function for the provision and monitoring of the vital portion of the job handling. It ensures that sufficient time is allocated for tasks. It also models the outputs of job processing.

VMCount is applied in the queuing phase to monitors the number of VMs provisioned for task processing.

InitReslist and *QueReslist* are the refinements for the provision and monitoring of the universes and functions required for the first and second phases of our model.

SystemState is a the refinement for *ProcessState*, *JobState*, and *JobTime* to reflect system state changes.

VMRequest is the refinement for the VM selection process during job queuing.

InitReqFunctions is the refinement for the provision and monitoring of the universes and functions required for the second phase (job initialising) of our model.

JobHandReslist is the refinement for the provision and monitoring of universes and functions required for the fourth phase of our model. It combines with a derived function called *jobhandling module* to process jobs.

Auto-scaler design is focused on optimizing metrics about the virtual infrastructure. Functions designed to model these metrics are described as follows.

TLevel defines the VMs threshold required for certain auto-scalers. The threshold could be t_{min}, t_{avg}, or t_{max} for minimum, average and maximum thresholds respectively.

VMUL defines VM utilization levels during job processing. The VM utilization levels could be $VMut_{min}$, $VMut_{avg}$, or $VMut_{max}$.

VMPool defines VM provisions in VM pools. Also, *VMPool* implements *RVM* to monitor VM optimisation levels for reusable VMs. The quantity of VMs in the pool could be q_{min}, q_{avg} or q_{max} for minimum, average and maximum quantities respectively.

VMPost defines VMs' position in the VI during job processing. VMs positions could be vm_f, vm_l for first and last positions which depicts the particular virtual machine that is being monitored by certain auto-scalers. These auto-scalers destroy VMs with less utilisation, unless the VM is the last one to be processed.

These *functions* and *universes* were combined to create the algorithms in our model.

3.4 Comparison between multimode and simple mechanisms

Our ASM model comprises of five *Transition Rules*. These rules are designed to reflect the execution phases an auto-scaler undergoes during job processing. The rules enable users to analyse the VM provision behaviours of auto-scaling mechanisms. We utilised algorithms to express the details of our *ground model* shown in figure 5. These algorithms were further refined according to the ASM refinement method into lower levels of abstractions. Our ground model and the refinements are later compared for equivalence according to *Börger's refinement* to check for the consistency of state transitions. The ASM Transition rules are: (i) Initial Phase (ii) Job Initialising (iii) Job Queuing (iv) Job Handling and (v) Job Termination.

The initial phase is the first transition rule of our model. This is the phase where all requisite resources are availed for job processing to commence. The system state is *idle* in this phase. The job initialising phase is the stage where job processing commences with the activation of system requests, and the mapping of jobs to VMs. The system state transitions to *active* in this phase. The job queuing phase is the stage where jobs queue due to the unavailability of VMs. The system state transitions to *waiting* in this phase. The job handling stages is the actual job processing stage where all the requisite resources are availed and jobs are processed till completion. The system state transitions to *busy* in this phase. The job termination phase is the final phase of our transition rules. This is the stage where all upload jobs are completely processed or a system interrupt causes job processing to halt. The system state transitions to either *done* or *stopped* in this phase.

The ASM *rules* are described in conjunction with algorithms 1 to 15 to identify the similarities existing between auto-scalers developed from different frameworks. Also, derived functions (ASM modules) were developed and applied to the *rules* of our model to ensure modularisation. Let us discuss our model's *rules* in the next sub-section.

JobState: Job → $\{idle, submitted, waiting, running, failed, done\}$
JobTime: Time → $\{idle, started, processing, stopped, completed\}$
ProcessState: Process → $\{new, ready, waiting, running, stopped\}$
SystemRequest: Request × AResource → $\{true, false\}$
SystemState: InfraState → $\{idle, active, waiting, busy, stopped, done\}$
JobOutcome: Job → $\{success, failure\}$
Compatible: Select(attr(j),attr(vm)) → $\{undef, true, false\}$
AddVM: VM × Job → $\{undef, true, false\}$
MappedJob: Job × VM → $\{undef, true, false\}$
MappedVM: Job × Process → $\{undef, true, false\}$
ReqResources: SystemRequest × AResource → $\{undef, true, false\}$
JobRequest: Job × AResource → $\{undef, true, false\}$
ProcessRequest:Process × AResource → $\{undef, true, false\}$
Event: Task → $\{start, aborted, terminated\}$
InitReslist: $IReslist$→ $\{IRL_{active}, IRL_{idle}, IRL_{busy}\}$
QueReslist: $QRlist$→ $\{QRL_{active}, QRL_{idle}, QRL_{busy}\}$
JobHandReslist: $JobhReslist$→ $\{JHRL_{active}, JHRL_{idle}, JHRL_{busy}\}$
InitReqFunctions: $InitReqFun$→ $\{IRF_{active}, IRF_{idle}, IRF_{busy}\}$
JobProcessing: $Jobproc$→ $\{JP_{active}, JP_{idle}, JP_{busy}\}$
Job: Process → Job
Jobhandler: Job → Joblauncher, Job → VM
Submitted: Job × VM → $\{undef, true, false\}$
BelongsTo: AResource × VM → $\{undef, true, false\}$
DestroyVm: VM → $\{undef, true, false\}$
ThresholdLevel: TLevel → $\{undef, T_{min}, T_{avg}, T_{max}\}$
VmPosition: VMPost → $\{undef, VM_F, VM_L\}$
ReusableVm: RVM → $\{undef, Q_{min}, Q_{avg}, Q_{max}\}$
VmPool: Vmpool→ $\{undef, Q_{min}, Q_{avg}, Q_{max}\}$
VMCount: NumofVMs→ $\{Num_{min}, Num_{avg}, Num_{max}\}$
VmUtilLevel: VmUL → $\{undef, UtVM_{min}, UtVM_{avg}, UtVM_{max}\}$
SimulationDuration: SimDur→ $\{SD_{min}, SD_{avg}, SD_{max}\}$
AveragQueTime: AvgQT → $\{AQT_{min}, AQT_{avg}, AQT_{max}\}$
AveragUtilPM: AvgUPM → $\{AUPM_{min}, AUPM_{avg}, AUPM_{max}\}$

Table 3: List of ASM functions

3.4.1 Rule 1, initial phase

Algorithms 1 and 2 depict the first phase of our model, for both *Simple* and *Multimode* auto-scalers as seen in figure 5. At this phase, all universes are provisioned for job processing to commence, however the process state is updated to *idle* as seen in lines 1 to 2 of algorithms 1 and 2. The system is inactive due to the absence of an ASM rule that initialises job processing. The *Simple* auto-scaler utilises the SimScaler (j,vm), BJS (j,vm) and GTP (j) to monitor jobs and VMs provisions; while the *Multimode* auto-scaler applies the VI (j, vm) and the joblauncher to do same as seen in lines 4. Also, jobs and process requests are activated but no response is received. Jobs are not submitted to VMs, but the VMs remain unmapped. The *Multimode* auto-scalers activate specific functions to monitor key indicators during auto-scaling as seen in lines 13 to 15 of algorithm 2. However, in the absence of job processing, they are all updated to *undef*. Algorithms 1 and 2 are refined in algorithm 3.

Initial phase refinement: The *InitReslist* function is utilised to check for the provision of universes for this phase. *InitReslist* is a refinement for all required universes and functions for this phase. *Systemstate* is updated to *idle* as seen in lines 2. Since there are no activities at this stage. *Systemrequests* and *ReqResources* are mapped to *false*, as jobs are not submitted to VMs. Also, jobs and VMs are not installed as tasks. This causes *systemstate* to remain updated to *idle*. This refinement is equivalent to our ground model's algorithms discussed in algorithms 1 and 2. As the state changes of algorithm 3 are equivalent to the state transitions of our model.

3.4.2 Rule 2, job initialising

The second phase of our model (shown in figure 5) begins with a system call activated by the application of an ASM *control state* rule. The universes provisioned from the previous phase are assigned. This transitions *processState* from *new* to *ready* as seen in lines 2 of our

Algorithm 1 *Simple Initial Phase*

1: **if** $\exists vm \in VM \wedge \exists p \in PROCESS \wedge \exists ar \in ARESOURCE \wedge \exists j \in JOB \wedge \exists t \in Time$ **then**
2: $processState(p) := idle$
3: **end if**
4: **while** $SimScaler(j, vm) \wedge BJS(j, vm) \wedge GTP(j)$ **do**
5: **if** $jobRequest(j, ar) = true \wedge processRequest(p, ar) = true$ **then**
6: $JobTime(j) := idle$
7: **end if**
8: **if** $mappedVM(j, p) = false \wedge mappedJob(j, p) = false$ **then**
9: $JobTime(j) := idle$
10: **end if**
11: **if** $installed(j, vm) = false \wedge Jobhandler(j, vm) = undef$ **then**
12: $JobState(j) := idle \wedge JobTime(j) := idle$
13: **end if**
14: **end while**

ground model algorithms 4 and 5. Specific universes are provisioned to monitor the activities of jobs and VM in line 4. The *Simple* auto-scalers utilise SimScaler (j,vm), BJS (j,vm) and GTP (j); while *Multimode* auto-scalers apply VI (j,vm) and the joblauncher. *JobRequests* and *processRequests* are activated to connect VMs to Jobs. *Jobtime* transitions to *started*. Jobs are mapped to VMs and installed as tasks as seen in lines 6 to 11. The *jobhandler* is activated to process the *tasks* in the *Simple* scalers whilst the *joblauncher* is activated for the *Multimode* auto-scalers. *Jobstate* is updated to *submitted* as seen in line 12. Algorithms 4 and 5 are refined in algorithm 6.

Job Initialising Refinement utilised the *InitReqFunctions* ASM derived function to check for the provisioning of requisite universes for this phase as seen in line 1 of algorithm 6. *InitReqFunctions* is a refinement for all required universes and functions for job initialising. The authentication of the universes updates *Systemstate* to *active*. *SystemRequest* is activated to initiate job requests and to provision VMs, which causes *systemstate* to be updated to *active* as seen in line 3 to 4. *ReqRequest* is applied to map jobs to VMs to be installed as tasks for the Jobhandler and Joblauncher to enforce their task processing. These activities cause the *Systemstate* to be updated to active as seen in line 7. This refinement is equivalent to the algorithms 4 and 5 of the second phase of our ground model as seen in figure 5. As the state changes of algorithm 6 are equivalent to the state transitions of our ground model.

3.4.3 Rule 3, job queuing:

The job queuing phase commences when there is a shortage of VMs during job processing. The phase is modelled as part of our ground model as seen in algorithms 7 and 8, and figure 5 for *Simple* and *Multimode* auto-scalers. In this phase, *universes* from the previous phases are provi-

Algorithm 2 *Multimode Initial Phase*

1: **if** $\exists vm \in VM \wedge \exists p \in PROCESS \wedge \exists ar \in ARESOURCE \wedge \exists j \in JOB \wedge \exists t \in Time$ **then**
2: $processState(p) := idle$
3: **end if**
4: **while** $VI(j, vm) \wedge Joblauncher(Jobhandler(j, vm))$ **do**
5: **if** $jobRequest(j, ar) = false \wedge processRequest(p, ar) = false$ **then**
6: $JobTime(j) := idle$
7: **end if**
8: **if** $mappedVM(j, p) = false \wedge mappedJob(j, p) = false$ **then**
9: $JobTime(j) := idle$
10: **end if**
11: **if** $installed(j, vm) = false \wedge Joblauncher(Jobhandler(j, vm), vm) = false$ **then**
12: $JobState(j) := idle \wedge JobTime(j) := idle$
13: $numofSerReq(s_i) := undef$
14: $VmUL = undef$
15: $Vmpool := undef \wedge RVM := undef$
16: **end if**
17: **end while**

Algorithm 3 *Refined Initial Phase*

Require: AResource
1: **if** $InitReslist = IRL_{idle} \wedge ReqResources = false$ **then**
2: $SystemState(j, p) := idle$
3: **end if**
4: **while** $SystemRequest = false \wedge ReqResources = false$ **do**
5: $SystemState(j, p) := idle$
6: **end while**
7: **if** $installed(j, vm) = false \wedge Jobhandler(j, vm) = undef$ **then**
8: $SystemState(j, p) := idle$
9: **end if**

sioned. This causes *jobTime* and *processState* to transitioned to *started* and *ready* as seen in lines 1 to 5. Job and process requests are activated to foster VM provisions. The VM count is monitored to determine the quantity of VMs available. When the quantity is below the required threshold for task processing; the *MappedVM*, *Jobhandler* and *joblauncher* are updated to *false* to confirm low VM count. This causes the jobs to queue as seen in lines 9 to 12 of both algorithms. *JobState* transitions to *waiting* to signify the current state of the modelling process. Our ground model algorithms are refined in algorithm 9.

Job Queuing Refinement is achieved by activating the *InitReqFunctions* and *ReqRequest* functions to foster resource provisions, and the mapping of VMs to jobs. This causes *SystemState* to be updated to *active* as seen in lines 1 to 2. The *QueReslist* derived function is applied to provision universes and functions for job queuing. Additionally, *SystemRequest* is activated to initiate jobs and

Algorithm 4 *Simple Jobs Initialising*

if $\exists vm \in VM \wedge \exists p \in PROCESS \wedge \exists ar \in ARESOURCE \wedge \exists j \in JOB \wedge \exists t \in Time$ **then**
  $processState(p) := ready$
end if
while $SimScaler(j, vm) \wedge BJS(j, vm) \wedge GTP(j)$ **do**
  if $jobRequest(j, ar) = true \wedge processRequest(p, ar) = true$ **then**
    $JobTime(j) := started$
  end if
  if $mappedVM(j, p) = true \wedge mappedJob(j, p) = true$ **then**
    $JobTime(j) := started$
  end if
  if $installed(j, vm) = true \wedge Jobhandler(j, vm) = true$ **then**
    $JobState(j) := submitted$
  end if
end while

Algorithm 5 *Multimode Jobs Initialising*

if $\exists vm \in VM \wedge \exists p \in PROCESS \wedge \exists ar \in ARESOURCE \wedge \exists j \in JOB \wedge \exists t \in Time$ **then**
  $processState(p) := ready$
end if
while $VI(j, vm) \wedge Joblauncher(Jobhandler(j, vm))$ **do**
  if $jobRequest(j, ar) = true \wedge processRequest(p, ar) = true$ **then**
    $JobTime(j) := started$
  end if
  if $mappedVM(j, p) = true \wedge mappedJob(j, p) = true$ **then**
    $JobTime(j) := started$
  end if
  if $installed(j, vm) = true$ **then**
    $JobState(j) := submitted$
  end if
end while

VMs requests as seen in lines 3 to 4. The VM count is monitored to check for the quantity of VMs. A reduction in the VM count causes *ReqRequest* and *Jobhandler* to be updated to *false*, which causes the *SystemState* to transition to *waiting* as seen in lines 5 to 7. This refinement is equivalent to our ground model algorithms, as their system states transitioned to *waiting* as seen in figure 5.

3.4.4 Rule 4, job handling

Job handling is the fourth phase of our model. This phase requires a derived function called the *Jobhandling Module* to optimise job processing, and the VM selection during job queuing. This function is created for each category of autoscalers (i.e. *Simple* and *Multimode Jobhandling* module) as seen in algorithms 10 and 11. These modules perform similar functions with different structural features. The *Simple Jobhandling* Module utilise *InitReqFunctions* and *QueRes-*

Algorithm 6 *Refined Job Initialising*

Require: AResource
if $InitReqFunctions = IRF_{active}$ **then**
  $SystemState(j, p) := active$
  while $SystemRequest = true \wedge ReqResources = true$ **do**
    $SystemState(j, p) := active$
  end while
  if $installed(j, vm) = true \wedge Jobhandler(j, vm) = true$ **then**
    $SystemState(j, p) := active$
  end if
end if

Algorithm 7 *Simple Jobs Queuing*

if $\exists j \in JOB \wedge \exists ar \in ARESOURCE \wedge \exists p \in PROCESS \wedge \exists vm \in VM \wedge \exists t \in Time$ **then**
  $JobTime(j) := started$
end if
while $SimScaler(j, vm) \wedge BJS(j, vm) \wedge GTP(j)$ **do**
  $processState(p) := ready$
  if $jobRequest(j, ar) = true \wedge processRequest(p, ar) = true$ **then**
    $JobTime(j) := started$
  end if
  if $VMCount \leq Num_{min}$ **then**
    $MappedVM(j, vm) := false$
    $Jobhandler(j, vm) := false$
    $JobState(j) := waiting$
  end if
end while

list to check for the provision for job initialising and queuing universes and functions. Once they are authenticated, *systemState* is updated to *active* as seen in lines 1 to 4. Autoscaler specific universes are provisioned to foster the exhibitions of VM provision behaviours. VMs and jobs requests are activated, which causes VM counts to be monitored. When the VM count falls below the required threshold, a system state change occurs. This causes *ReqResources* and *Jobhandler* to be updated to *false*. The *systemState* transitions to *waiting* as seen in lines 5 to 10 of both algorithms. The *ReqResources* is rechecked periodically, to confirm if jobs have been mapped to VMs and installed as tasks. If the response is negative, the VM selection mode is activated via *AddVM* function. Once VM selection is accomplished, jobs are then mapped to VMs as seen in lines 11 to 19 of algorithm 10. This process differs from the *Multimode* Jobhandling module. The *VmRequest* function (a refinement of the VM selection process) is activated, which causes jobs to be mapped to VMs via the *ReqResources* as seen in lines 12 to 16 of algorithm 11.

Job handling commences with the application of *InitReqFunctions* to foster the provision of job initialising universes and functions. Once these are *active*, the *JobhandMod* is activated to provision all the universes and func-

Algorithm 8 *Multimode Jobs Queuing*

1: **if** $\exists j \in JOB \wedge \exists ar \in ARESOURCE \wedge \exists p \in PROCESS \wedge \exists vm \in VM \wedge \exists t \in Time$ **then**
2: $JobTime(j) := started$
3: **end if**
4: **while** $VI(j, vm)$ **do**
5: $processState(p) := ready$
6: **if** $jobRequest(j, ar) = true \wedge processRequest(p, ar) = true$ **then**
7: $JobTime(j) := started$
8: **end if**
9: **if** $VMCount \leq Num_{min}$ **then**
10: $MappedVM(j, vm) := false$
11: $Joblauncher(Jobhandler(j, vm)) := false$
12: $JobState(j) := waiting$
13: **end if**
14: **end while**

Algorithm 9 *Refined Jobs Queuing*

Require: AResource
1: **while** $InitReqFunctions = IRF_{active} \wedge ReqResources = true$ **do**
2: $SystemState(j, p) := active$
3: **if** $QueReslist = QRL_{active} \wedge SystemRequest(p, ar) = true$ **then**
4: $SystemState(j, p) := active$
5: **if** $VMCount \leq Num_{min}$ **then**
6: $ReqResources := false$
7: $Jobhandler((j, vm) := false$
8: $SystemState(j, p) := waiting$
9: **end if**
10: **end if**
11: **end while**

tions for job handling. This causes *SystemState* to be updated to *active* as seen in lines 1 to 3 of algorithm 12. Sufficient time request is made and the response granted by the *AResources* to ensure that the jobs provisioned are adequately processed. The *SystemRequest* is activated to foster jobs and process requests. An authentication of this request, maps jobs to VMs via *ReqResources*. This causes the mapped jobs and VMs to be installs as tasks to either continue or restart job processing as seen in lines 5 to 10. The outputs of job processing are monitored with *SimulionDuration*, *AverageUtilPM* and *AverageQueTime*. The job processing activity causes the *SystemState* to transition to *busy* as seen in lines 11 to 17. Our ground model's job handling algorithms are refined in algorithm 13.

Job Handling Refinement is accomplished by activating the the *InitReqFunctions* and *SystemRequest* functions to foster the provision of the requisite universes for job initialising. This causes *SystemState* to be updated to *active* as seen in line 1 of algorithm 13. Moreover, the *Jobhandling Module* and the *jobhandler* are activated to foster VMs selection during job queuing, and the installing of mapped VMs and jobs as tasks for job processing as seen in lines 2 to 3 . The *JobProcessing* function is activated to foster time

Algorithm 10 *Simple Jobhandling Module*

Require: AResource
1: **while** $InitReqFunctions = IRF_{active}$ **do**
2: $SystemState(j, p) := active$
3: **if** $QueReslist = QRL_{active} \wedge SystemRequest(p, ar) = true$ **then**
4: $SystemState(j, p) := active$
5: **end if**
6: **while** $SimScaler(j, vm) \wedge BJS(j, vm) \wedge GTP(j)$ **do**
7: **if** $VMCount \leq Num_{min}$ **then**
8: $ReqResources := false$
9: $Jobhandler((j, vm) := false$
10: $SystemState(j, p) := waiting$
11: **end if**
12: **if** $(ReqResources = false \wedge installed(task(j, vm)) = false$ **then**
13: $AddVM(vm, j) := true$
14: $Compatible(attr(j), attr(vm)) := true$
15: $belongsTo(j, vm) := true$
16: $ReqResources := true$
17: **end if**
18: **end while**
19: **end while**

requests and the mapping of jobs to VMs for job processing. Furthermore, the output of job handling are modelled as the job processing ensues. This activity causes the *SystemState* to transition to *busy* as seen in lines 5 to 12. This refinement is equivalent to our ground model's job handling algorithm, as *SystemState* transitioned to *busy* as seen in figure 5.

3.4.5 Rule 5, job termination

Job Termination is the fifth phase of our model. This phase requires two conditions to be initiated. First, a system failure or an abrupt system call to halt job processing. Second, the exhaustion of jobs generated. This phase is a result of the activities of job handling as seen in algorithm 14 of our ground model. Hence, there must be activities ongoing to demonstrate job processing, before job termination occurs. Therefore, before job processing is halted, the *universes* and *functions* required for job initialising should be provisioned. This updates *processState* to *ready* as seen in lines 1 to 2. Job processing is monitored via the allocated time, and how long jobs remain mapped to VMs. As job are being processed, *processState* and *jobState* transtion to *running*. Also *jobTimes* updates to *processing* as seen in lines 3 to 10. When a system interrupt occurs which causes the job processing to halt; *jobstate* transitions to *failed* and *processState* to *stopped* as seen in lines 13 to 15. Moreover, when job processing is completed; *jobState* transitions to *done*, *jobTime* to *completed*, and *event(t)* to *terminate* as seen in lines 17 to 20 depicting the exhaustion of job generated. The system state changes can be seen in the job and process states figures 6 and 7. Job termination is refined in algorithm 15.

Job Termination Refinement is achieved by the ap-

Algorithm 11 *Multimode Jobhandling Module*

Require: AResource
1: **while** $InitReqFunctions = IRF_{active}$ **do**
2: $SystemState(j,p) := active$
3: **if** $QueReslist = QRL_{active} \wedge SystemRequest(p,ar) = true$ **then**
4: $SystemState(j,p) := active$
5: **end if**
6: **while** $VI(j,vm)$ **do**
7: **if** $VMCount \leq Num_{min}$ **then**
8: $ReqResources := false$
9: $Joblauncher(Jobhandler(j,vm),vm) := false$
10: $SystemState(j,p) := waiting$
11: **end if**
12: **while** $VmRequest(j,vm) = active$ **do**
13: $ReqResources := true$
14: **end while**
15: **end while**
16: **end while**

Algorithm 12 *Job Handling*

Require: AResource
1: **while** $InitReqFunctions = IRF_{active} \wedge SystemRequest(p,ar) := true$ **do**
2: $JobHandMod_{multimode} := active$
3: $Jobhandler((j,vm) := true$
4: $SystemState(j,p) := active$
5: **while** $t \in TIME \wedge TimeRequest(j,p) = true$ **do**
6: $mappedVM(j,vm) := true$
7: **if** $SystemRequest(p,ar) = true$ **then**
8: $ReqResources := true$
9: $installed(j,vm) := true$
10: $event(t) := started$
11: $SimulationDuration := SD_{max}$
12: $AveragUtilPM := AUPM_{max}$
13: $AveragQueTime := AQT_{max}$
14: **end if**
15: $SystemState(j,p) := busy$
16: **end while**
17: **end while**

Algorithm 13 *Refined Job Handling*

Require: AResource
1: **while** $InitReqFunctions = IRF_{active} \wedge SystemRequest(p,ar) := true$ **do**
2: $JobHandMod_{multimode} := active$
3: $Jobhandler((j,vm) := true$
4: $SystemState(j,p) := active$
5: **while** $JobProcessing = JP_{active}$ **do**
6: $SystemRequest(p,ar) := true$
7: $ReqResources := true$
8: $SystemState(j,p) := busy$
9: **end while**
10: **end while**

plication of *InitReqFunctions*, *SystemRequest* and *ReqResources*. This is to foster the provision of *universes* and *functions* for job initialising and job handling. This causes *systemstate* to transition to *active* as seen in lines 1 to 2 of algorithm 15. The *JobHandReslist* function and the *jobhandler* are activated to foster job processing. *Systemstate* is updated to *busy* as seen in lines 3 to 4. When a system call is activated that causes job processing to halt; the *systemstate* is automatically updated to *stopped* as seen in lines 5 to 6. This event signifies an abrupt job termination. Furthermore, when *systemRequest* is *active* while there are no jobs to be processed; *event* and *systemstate* are updated to *terminate* and *done* as seen in lines 7 to 12. This signifies the completion of job processing. This refinement is equivalent to our ground model's algorithm 14 as *systemstate* transitioned to either *stopped* or *done* aligning with the job termination conditions. Let us now discuss the evaluation of our model.

4 Evaluation

This section is split into two parts: the validation and evaluation of our model. The validation sub-section focuses on test cases developed from our model's previously discussed algorithms. The evaluation sub-section focuses on examining the five auto-scalers used to construct our model. Finally, we conclude our evaluation by demonstrating our model's applicability to other auto-scalers from prior art. Let us now proceed to discuss the validation of our model.

4.1 Validation

Our ASM model's validation was achieved via the creation of test cases from our ground model's algorithms and their refinements. These test case were developed on CoreASM (a plug-in available for the Eclipse IDE). The test cases (i.e., *coreasm specifications*) were designed as interactive sequences with suitable checks to describe the expectations of our model's states. These test cases were checked to see if specified assertions hold in given states. The test cases were processed and checked if all the assertions were satisfied. A satisfied assertion finished with a *pass* verdict. However, as soon as an assertion was not satisfied, the simulation was interrupted, reporting a violation. At each step, the simulator performed update checking to ensure that all states were updated. Our validation goals are:

To assess the interactions of the phases of our ground model via the application of universes and signatures.

To examine the application of derived functions (modules) as refinements of our ground model.

To assess the application of *guarded updates* (which are reflective of control state ASMs) to ensure equivalence (between ground model algorithms and their refinements).

Our validated specifications are available in the Auto-

Algorithm 14 *Job Termination*

if $\exists j \in JOB \wedge \exists ar \in ARESOURCE \wedge \exists p \in PROCESS \wedge \exists t \in Time \wedge JobRequest(j, ar)$ **then**
  $processState := ready$
  while $t \in TIME \wedge TimeRequest(j, p) = true \wedge mappedVM(j, vm) = true$ **do**
    $JobTime(j) := processing$
    while $processRequest(p, ar) = true \wedge jobhandler(j, vm) = true$ **do**
      $installed(j, vm) := true$
      $event(t) := started$
      $JobState(j) := running$
      $ProcessState(p) := running$
      $JobTime(j) := processing$
    end while
  end while
  if $jobRequest(j, ar) = true \wedge event(task(p)) = terminate$ **then**
    $JobState(j) := failed \wedge processState(p) := stopped$
  end if
end if
if $\neg\exists j \in JOB \wedge \neg\exists p \in PROCESS \wedge JobRequest(j, ar)$ **then**
  $Event(t) := Terminate \wedge jobTime(j) := Completed$
  $jobState(j) := done$
end if

Algorithm 15 *Refined Job Termination*

Require: AResource
while $InitReqFunctions = IRF_{active} \wedge SystemRequest(p, ar) = true$ **do**
  $ReqResources := true \wedge SystemState(j, p) := active$
  while $JobHandReslist = JHRL_{active} \wedge jobhandler(j, vm) = true$ **do**
    $SystemState(j, p) := busy$
    if $SystemState(j, p) = busy \wedge event(task(p)) = terminate$ **then**
      $SystemState(j, p) := stopped$
    else if $SystemRequest(p, ar) \wedge \neg\exists j \in JOB \wedge \neg\exists p \in PROCESS$ **then**
      $Event(t) := Terminate$
      $SystemState(j, p) := done$
    end if
  end while
end while

Scaling-ASM repository on Github[2]. This allows for further scrutiny and reuse of our validation approach. We provide a short overview of the validation process.

4.1.1 Model modules

CoreASM modules (CoreModules) were designed with ASM rules which aligned with the control state ASM definition 1 to ensure that, once the conditions for state transitions were met, system states were updated accordingly.

Three key modules (shown in figure 8) were applied to support our model validation. These modules represents the actual file names of the modules with the ASM specification file extensions (.casm). The arrows represents the levels of dependency of each file from the bottom to the top. The topmost file is the module for job initialising, which is followed by the job queuing module. The *jobhandler module* for *Simple* and *Multimode* auto-scalers are on the same level, which are followed by a general *jobhandler module* with VM selection integrated refinements. The *MultiJobHandMod* and *SimJobHandMod* modules were validated separately, and applied to the *Multimodes* and *Simple* auto-scalers as seen in MultiJobHandMod.casm and SimJobHandMod.casm.

In order to validate the modules, the test case were checked for the provision of *aresources* and job queuing resources via the *InitReqFunctions* and *QueReslist* functions. It checked for the provisions of specific auto-scaling universes for the two categories of auto-scalers via the *JobInitFunMod.casm* module file. The result showed a *check succeeded* assertion verdict with *systemstate* updated to *active*. Moreover, the test case checked for the number of VMs available via the *VMcount*. The result showed a *check succeeded* assertion. The *Jobhandler*, *Joblauncher* and *ReqResource* were updated to *false* which caused *systemstate* to transition to *waiting*. The test case checked for VM selection. The result showed a *check succeeded* assertion. The *AddVM* function was activated to select VMs which caused *ReqResource* to transition to *true*. The two specialized modules fostered the transition of their auto-scalers from queuing to job handling. The CoreModules were then utilised to validate the other test cases representing the phases of our ground model.

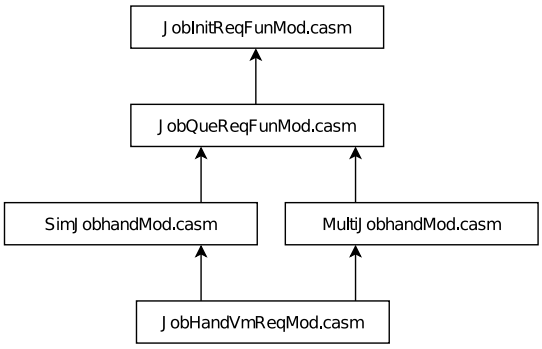

Figure 8: ASM validation modules

Initial Phase was validated following rule 1 (discussed in sub-section 3.4.1) of our model as seen in the *JobInitFunMod.casm* file. A series of checks were done to validate this phase. First, the test case checked for the provision of *universes* and *functions*. The result showed a *check succeeded* assertion. The *processState* function was updated to *ready*. Second, the test case checked for jobs and process

[2] https://github.com/EbenezerKomlaGavua/Auto-Scaling-ASM

requests via *jobRequest* and processRequest functions. An expected *false* result was returned with a *check succeeded* assertion. A false result was expected because no requests is made at the initial phase. Third, a *mappedVM* checks were done and an expected *false* result was returned with a a *check succeeded* assertion. Finally, the test case checked if VMs were mapped to jobs and installed as tasks. The result showed a *check succeeded* assertion. The above validation process can be seen in the SimpleInitialPhase.casm and MultimodeInitialPhase.casm files.

Moreover, the test cases for the refinements of the initial phase were checked to ensure equivalence with those representing our ground model. The CoreASM checked for the provision of universe and functions via *InitReslist* function. The results showed a *check succeeded* assertion. *Systemstate* was updated to *active*. The test case checked for job and process requests via *SystemRequest*. The result showed a *check succeeded* assertion. The test case then checked for the mapping of VMs and jobs via *ReqResource*. The result showed a *check succeeded* assertion. Finally, the test case checked the status of *joblauncher* and *jobhandler*. The result showed a *check succeeded* assertion with an expected *false*, which showed that jobs and VMs were not installed as tasks. The checks on the refinement were equivalent to those of our ground model. The above validation process can be seen in the RefinedInitialPhase.casm file.

Job Initialising was validated following rule 2 (discussed in sub-section 3.4.2) of our model. The *Simple* and *Multimode* auto-scalers were validated separately. The test case for job initialising checked for the provision and interaction of universes and functions from the previous phase via the *InitReqFunctions* function in the *JobInitFunMod.casm* module file. Also, the test case checked for the application of an ASM rule to initiate job processing and the request for jobs and processes. The result showed a *check succeeded* assertion verdict with *jobTime* updated to *started*. The test case checked for the installation of mapped jobs as tasks. The result showed a *check succeeded* assertion with *jobTime* updated to *started* and *jobState* to *submitted*. The job initialising validation can be seen in the SimpleJobJobInitialising.casm and MultimodeInitialising.casm files.

Furthermore, the test case for the refined algorithms checked for the provision of universes and functions via *InitReqFunctions*. The result showed a *check succeeded* assertion verdict with *systemstate* updated to *active*. The test case checked for jobs and VMs request via *systemRequest*. A *check succeeded* assertion verdict was returned. Also, the test case checked for the installation of mapped jobs as tasks. The result showed a *check succeeded* assertion verdict with the *jobhandler* updated to *true*. The *systemstate* was updated to *active* which was equivalent to the checks executed on the test cases of our ground model. The refined job initialising validation can be seen in the RefinedJobInitialising.casm file.

Job Queuing was validated per rule 3 of our model (discussed in sub-section 3.4.3). The *Simple* and *Multimode* auto-scalers were validated separately. The test cases checked for the provision of of universes from the first two phase of our model. It also checked for specific auto-scalers universes and functions as seen in the *JobQueFunMod.casm* module file. The result showed a *check succeeded* assertion. *JobTime* and *processState* were updated to *started* and *ready*. The test cases checked for jobs and VMs requests and the VM count. The result showed a *check succeeded* assertion. When the VM count fell below the expected threshold, the *jobhandler* was updated to *false* in the *Simple* auto-scalers and the *joblauncher* was also updated to *false* in *Multimode* auto-scalers. This situation transitioned *JobState* to *waiting*. The job queuing validation can be seen in the SimpleJobQue.casm and MultimodeJobQue.casm files.

Moreover, the test case for the refined job queuing was validated. The test case checked for the provision of universes and functions via the *QueReslist* function. The result showed a *check succeeded* assertion which caused *systemstate* to be updated to *active*. Also, the test case checked for for jobs and VMs requests, and the level of the VM count. The result showed a *check succeeded* assertion which caused *systemstate* to be updated to *waiting*. This showed that there was shortage of VMs. The test cases demonstrated the equivalence of the refined queuing algorithms to those of our ground model. The refined job queuing validation can be seen in the RefinedJobQue.casm file.

Job Handling The test case created to validate Job handling applied rule 4 of our model (discussed in subsection 3.4.4). This phase applied the *MultiJobHandMod* and *SimJobHandMod* modules discussed in section 4.1.1 to validate the two categories of auto-scalers. The test case checked for the provision and interaction of *universes* and *functions* via *InitReqFunctions* in the *JobInitFunMod.casm* file. Also, the test case checked for the job queuing and VM selection via the *jobhandling modules* in *JobHandVmReqMod.casm* file. The results showed a *check succeeded* assertion. Moreover, the test cases checked for the request for sufficient time for tasks processing via *timeRequest* and the installation of tasks. The results showed a *check succeeded* assertion. The *systemstate* transitioned to *busy*. The Job handling validation can be seen in the JobHandling.casm file.

Furthermore, the refined algorithm was also validated with the *Jobhandling module*. The test cases checked for the provision resources via *JobInitFunMod.casm* file. Also, it checked for the VMs shortage and selection at the queuing phase with the *jobhandling module*. The results showed a *check succeeded* assertion which caused the *systemstate* to transition to *active*. The test case checked for the mapping VMs to jobs, time requests and *systemrequest* via the *JobProcessing* function. The results showed a *check succeeded* assertion. The refined job handling test cases showed equivalence to those our ground model, since they all caused a system state update to *busy*. The refined Job handling validation can be seen in the RefinedJobHan-

dling.casm file.

Job Termination was validated following subsection 3.4.5 of our model. The test cases developed for this phase checked the conditions for job termination. Sequential checks were employed to ensure that all the previous phases were appropriately validated. First, the test case checked for job initialising via the *InitReqFunctions* function in the *JobInitFunMod.casm* file. The results showed a *check succeeded* assertion which caused *processstate* to transition to *ready*. Second, the test case checked for the mapping of VMs to jobs, time request and the status of the *Jobhandler*. The results showed a *check succeeded* assertion. Third, the test case checked for system interruptions (via universes designed to halt midway during task processing). The results showed a *check succeeded* assertion. Fourth, the test case checked for the quantity of jobs available at the end of task processing. The results showed a *check succeeded* assertion. The validation can be seen in the JobTermination.casm file.

Job Termination refinement test case checked for universes and functions via the *InitReqFunctions* function. The result showed a *check succeeded* assertion which caused *systemstate* to transition to *active*. Then, the test case checked for job handling via *JobHandRelist*. A *check succeeded* assertion was returned and *systemstate* to transition to *busy*. Moreover, the test case checked for system state interruption. A *check succeeded* assertion was returned and *systemstate* transitioned to *stopped* (as expected). Finally, the test case checked for *systemRequest* when all the jobs generated were exhausted. A *check succeeded* assertion was returned, which caused *systemstate* to transitioned to *done* (as expected). The refined job termination test cases showed equivalence to those of our ground model, since they all caused a system state transition to *stopped* and *done* per the conditions for job termination. The validation process can be seen in the RefinedJobTermination.casm file.

In conclusion, the test cases developed from our ground model algorithms and their refinements aligned with the validation goals discussed in section 4.1.

4.2 Abstract state machine model evaluation

This section describes the evaluation process of our model. Our evaluation criteria are discussed emphasizing the ASM theory described in sub-section 2.4. We applied the following criteria to evaluate our model.

- To assess the equivalence of the refined auto-scaler algorithms to our ground model via the application of *universes* and *signatures*.
- To examine the application of derived functions to specific portions of auto-scaling; such as job initialising, VM selection and job handling to ensure the application of *ASM Model Refinement*.
- To assess the application of *guarded updates* and *Börger's refinement* on auto-scalers.

Algorithm 16 $Threshold\ \ Initial\ \ Phase$

Require: AResource
1: **if** $InitReslist = IRL_{idle}$ **then**
2: $TLevel := undef \wedge SystemState(j,p) := idle$
3: **end if**
4: **while** $SystemRequest = false$ **do**
5: $SystemState(j,p) := idle$
6: **end while**
7: **if** $ReqResources = false$ **then**
8: $SystemState(j,p) := idle$
9: **end if**
10: **if** $installed(j,vm) = false$ **then**
11: $Joblauncher(Jobhandler(j,vm)) := false$
12: $SystemState(j,p) := idle$
13: **end if**

In order to prevent the repetition of algorithms, we will selectively utilise a few formalized algorithms for our discussion. Let us discuss the evaluation of our ASM rules.

4.2.1 Rule 1, initial phase

This phase was evaluated per sub-section 3.4.1 and algorithm 1 of our model. Algorithm 16 is used for our discussion, since aside the specific function *TLevel*, the state changes are the same for all auto-scalers. At the initial phase, all the auto-scalers apply the *InitReslist* function to access universes and functions. However, no *aresources* are provisioned. Therefore *systemstate* is updated to *false* and the threshold monitoring function *TLevel* is updated to *undef* as seen in lines 1 to 3 of algorithm 16. Also, job and process requests are initiated, but no responses are received (as expected). Hence, there was no state transitions for *systemstate* as seen in lines 4 to 6. Also, no VMs were mapped to jobs, and installed as tasks for processing. This caused the *Joblauncher* to be updated to *false* and system state to *idle* as as seen in lines 7 to 13.

The refinement is equivalent to the first phase of our ground model shown in figure 5. Since the system state transitioned to *idle* as seen in algorithm 3. Also, the refinement satisfies the evaluation criteria discussed in subsection 4.2, since derived function were applied to model the refinement. Also, state transitions were observed when conditions were met, which are reflective of *guarded updates* of control state ASMs.

4.2.2 Rule 2, job initialising

Job Initialising was evaluated with sub-section 3.4.2. Algorithm 17 is utilised for our discussion, since all the state changes are the same for all auto-scalers (aside the specific reusable VMs function *RVM*). The *InitReqFunctions* derived function is applied to foster the provision of *Aresources* via universes and functions for this phase. The *SystemRequests* and *ReqResources* functions activated jobs and VMs requests, and the mapping of VMs to jobs as seen

Algorithm 17 $Vmopt\ \ Jobs\ \ Initializing$

Require: AResource
1: **if** $InitReqFunctions = IRF_{active} \wedge SystemRequest = true$ **then**
2: $Joblauncher(Jobhandler(j, vm)) := true \wedge ReqResources := true$
3: **if** $installed(j, vm) = true \wedge RVM = Q_{min}$ **then**
4: $SystemState(j, p) := active$
5: **end if**
6: **end if**

Algorithm 18 $Vmcreate\ \ Jobs\ \ Queuing$

Require: AResource
1: **while** $InitReqFunctions = IRF_{active}$ **do**
2: $ReqResources := true \wedge TLevel := T_{min}$
3: $SystemState(j, p) := active$
4: **if** $QueReslist = QRL_{active}$ **then**
5: $SystemState(j, p) := active$
6: **end if**
7: **if** $VMCount \leq Num_{min}$ **then**
8: $ReqResources := false$
9: $Joblauncher(Jobhandler(j, vm)) := false$
10: $TLevel := T_{avg}$
11: $SystemState(j, p) := waiting$
12: **end if**
13: **end while**

in lines 1 to 2. Tasks are installed for job processing to commence and the function *RVM* is updated to minimum, which caused *SystemState* to transition to *active* as seen in lines 3 to 6. This evaluation is applicable to *Threshold*, *Vmcreate*, *Pooling* and *FixedVM* auto-scalers. In the case of the other *multimode* auto-scalers, *threshold* and *vmcreate* transition to T_{min}, while *pooling* to Q_{min} during job initialising.

This refinement is equivalent to the second phase of our model shown in figure 5 and algorithm 6. Also, the refinement satisfies our evaluation criteria discussed in subsection 4.2, since derived function were applied to cause job processing to commence. State transitions were seen when conditions were met, which are reflective of *guarded updates* of control state ASMs.

4.2.3 Rule 3, job queuing

Job queuing evaluation was achieved via Rule 3 of our model discussed in sub-section 3.4.3. Algorithm 18 (representing Vmcreate auto-scaler) is used for our discussion. The *InitReqFunctions* function is activated for the provision of *universes* to foster job initialising. This caused *systemState* to transition to *active*. *Tlevel* transitions to T_{min} (i.e., minimum VM threshold utilisation) as seen in lines 1 to 3. *QueResList* is activated to monitor process and job requests, and the mapping of VMs to jobs as seen in lines 4 to 6. The VM count are monitored. A reduction in the VM count, caused *ReqResources* and *Joblauncher* to be updated to *false*. Also, *Tlevel* transitions to T_{avg} (i.e., average VM threshold utilisation). This caused *SystemState* to transition to *waiting* as seen in lines 7 to 13. This evaluation is applicable to all auto-scalers. In the case of the other *multimode* auto-scalers *Threshold* transitioned to T_{avg}, *Pooling* to Q_{avg} and *Vmopt* to Q_{avg} during job queuing.

This refinement is equivalent to the job queuing of our model shown in figure 5 and algorithm 9. Also, the refinement satisfies the evaluation criteria discussed in subsection 4.2. This is seen in the application of derived functions to model job queuing. Also, state changes are seen when function conditions were met, which are reflective of the *guarded updates* of control state ASMs.

4.2.4 Rule 4, job handling

Job handling was evaluated per Rule 4 (discussed in subsection 3.4.4). During job handling, the auto-scalers activate the *InitRequiredFunctions*, *JobHandRelist* derived functions and the *jobhandling modules* (designed in algorithms 10 and 11) for job processing. Aresources are provisioned via universes and functions for job processing. This caused *SystemState* to transition to *busy* as seen in lines 1 to 6 of algorithm 19.

The auto-scalers exhibited behaviours per their core functions. The behaviours are analysed as follows. First, *threshold* and *vmcreate* applied *VmUL* to monitor VM utilization. *VmUL* is utilised differently in the two auto-scalers. In the case of *threshold*, if the current VM utilisation is lower than the average VM threshold, the VMs are destroyed. However, if the current VM utilisation is lower than the average VM threshold but the VM is the last VM being processed; the duration period is extended by one hour to receive a new job before the VM is destroyed. In the case of *Vmcreate*, if the maximum utilisation of VMs is greater than the expected VM threshold, more VMs added are created. Also, if the current threshold is greater than the expected VM threshold, more VMs are created. This caused *SystemState* to transition to *busy*.

Second, *Vmopt* monitors the VM count of reusable VMs in the VI. If the number reusable VMs are more than and equal to the minimum number expected, job processing continues until all the jobs are processed. Third, *Pooling* monitors the VM count in its VM pool during job processing. If the number of VMs are more that the minimum expected, more VMs are created. Job processing continues until all the jobs are processed as seen in lines 7 to 12 of algorithms 19. This causes *SystemState* to transition to *busy*. The output of job processing showed *SimulationDuration*, *AveragQueTime* and *AveragUtilPM* transitioned to *maximum* levels.

These job handling refinements are equivalent to the algorithms of our ground model shown 12 to 13 which are reflective of figure 5. Also, the refinement satisfies the evaluation criteria discussed in sub-section 4.2 as derived functions were applied to model job initialising to job handling.

Algorithm 19 *Pooling Jobs handling*

Require: AResource
1: **while** $InitReqFunctions = IRF_{active}$ **do**
2: $JobHandMOD_{multimode} := active$
3: **if** $JobHandReslist = JHRL_{active} \wedge Vmpool = Q_{max}$ **then**
4: $ReqResources := true \wedge$
5: $SystemState(j,p) := busy$
6: **end if**
7: **if** $Vmpool >= Min_{\mathrm{Q}}$ **then**
8: $AddVM(j,vm) := true \wedge$
9: $ReqResources := true$
10: $Joblauncher(Jobhandler(j,vm)) := true$
11: $SystemState(j,p) := busy$
12: $SimulationDuration := SD_{max}$
13: $AveragQueTime := AQT_{max}$
14: $AveragUtilPM := AUPM_{max}$
15: **end if**
16: **end while**

4.2.5 Rule 5, job termination:

Job Termination was evaluated per rule 5 of our model (discussed in sub-section 3.4.5). The *pooling* auto-scaler was utilised to evaluate this phase. The evaluation of job termination required a modelling that showed an interaction of the previously discussed phases.

Initially, *InitRequiredFunctions*, *jobhandling modules* are activated for the provision of *Aresources* via universes and functions towards job initialising and job handling. The *JobHandRelist* is triggered to foster the job handling. *Vmpool* transitions to Q_{max} to signifying sufficient provision of VMs for job processing. *SystemState* transitions to *busy* as seen in lines 1 to 4 of algorithm 20. Moreover, the VM count is checked in the VM pool. If the number of VMs in the VM pool is within the minimum range, more VM are provisioned. This causes the *systemState* to transition to *busy* as seen in lines 7 to 10. The *SimulatioDuration* transitions to *SDmax* and *AverageQueTime* to *AQTmax*. Also *AverageUtilPM* transitions to *AUPMmax* as seen in lines 11 to 13. Furthermore, while jobs are being processed, a terminate event causes *systemState* transitions to *stopped* and *event* to *terminate* as seen in lines 15 to 16. Also, when the jobs generated are exhausted while *SystemRequest* is activated; *event* transitions to *terminate* and *systemstate* to *done*as seen in lines 17 to 21.

This job termination refinement of the *pooling* auto-scaler is equivalent to the fifth phase of our model shown in figure 5, and algorithm 14 to 15. Also, the refinement satisfies the evaluation criteria discussed in sub-section 4.2 as derived functions were utililsed to model job initialising to job termination. Also, state changes were seen when the conditions for function were met, which are reflective of *guarded updates* of control state ASMs. Also, there is interaction between the phases of our model via the application of universes and functions.

In conclusion, this evaluation enabled us to check the applicability of our model to the auto-scalers offered with DISSECT-CF, and also the flexibility of the transition rules of our model.

Algorithm 20 *Pooling Jobs Termination*

Require: AResource
1: **while** $InitReqFunctions = IRF_{active}$ **do**
2: $JobHandMOD_{multimode} := active$
3: **if** $JobHandReslist = JHRL_{active} \wedge Vmpool = Q_{max}$ **then**
4: $ReqResources := true$
5: $SystemState(j,p) := busy$
6: **end if**
7: **if** $Vmpool >= Min_{\mathrm{Q}}$ **then**
8: $AddVM(j,vm) := true$
9: $ReqResources := true$
10: $SystemState(j,p) := busy$
11: $SimulationDuration := SD_{max}$
12: $AveragQueTime := AQT_{max}$
13: $AveragUtilPM := AUPM_{max}$
14: **end if**
15: **if** $SystemState(j,p) = busy \wedge event(task(p)) = terminate$ **then**
16: $SystemState(j,p) := stopped$
17: **else if** $SystemRequest(p,ar) \wedge \neg\exists j \in JOB \wedge \neg\exists p \in PROCESS$ **then**
18: $Event(t) := Terminate$
19: $SystemState(j,p) := done$
20: **end if**
21: **end while**

4.3 Evaluation of our model with another auto-scaling mechanism

The literature in section 2 analysed past auto-scalers mechanisms, and selected [7] for in-depth analysis with our model. The algorithms of Yang et al.'s work have been made public; hence it was possible to apply our model. This auto-scaler presents a typical auto-scaling approach using workload prediction, as well as horizontal and vertical scaling. Therefore, it was possible to analyse and classified it as a *multimode* auto-scaler due to its specific features. We discuss only the job handling phase since the other phases apply features similar to those discussed per the *multimode* auto-scalers above.

Job Handling: In order to evaluate this auto-scaler, the *jobhandling module* for *multimode* auto-scalers is adopted and applied to optimize VM selection. VM selection was activated via *VmRequest*. This caused VMs and jobs to be mapped and installed as tasks to continue job processing.

The auto-scaler by [7] applies *InitReqFunctions* to initiate job processing. This activates the Workload prediction function and *jobhandling module*. This causes *systemstate* to be updated to *active* as seen in lines 1 to 5 of algorithm 21.

JobHandReslist is activated to foster the provision of resources via universes and functions for job handling. This activity causes the VMs utilisation levels to in-

Algorithm 21 *LoadPredict Jobs handling*

Require: AResource
1: **if** $InitReqFunctions = IRF_{active} \wedge SystemRequest(p, ar) = true$ **then**
2: $WorkloadPrediction := PWL_{active} \wedge ReqResources := true$
3: $JobHandMod_L := active \wedge Joblauncher(Jobhandler(j, vm)) := true$
4: $SystemState(j, p) := active$
5: **end if**
6: **while** $JobHandReslist = JHRL_{active}$ **do**
7: **if** $VmUL_t > VMut_{max}$ **then**
8: $SystemState(j, p) := busy$
9: **end if**
10: **for all** $vm \in VM$ **do**
11: **if** $VmUL_t > VMut_{max}$ **then**
12: $selfhealing_{SU} := active$
13: **end if**
14: **if** $VmUL_t > VMut_{max}$ **then**
15: $AR_{SU} := active$
16: **end if**
17: **if** $VmUL_t < VMut_{min}$ **then**
18: $AR(VR)_{SD} := active$
19: **end if**
20: **end for**
21: $AveragUtilPM := AQT_{max}$
22: $SimulationDuration := SD_{max}$
23: $AveragQueTime := AUT_{max}$
24: $SystemState(j, p) := busy$
25: **end while**

Algorithm 22 *Virtual Resource scaling down*

Require: AResource
1: **while** $JobHandMod_L = busy$ **do**
2: $SystemState(j, p) := busy$
3: **while** $JobHandReslist = JHRL_{active}$ **do**
4: **if** $VmUL_t < VMut_{max}$ **then**
5: $VMLevel_{SD} := active$
6: **end if**
7: **if** $VmUL_t < VMut_{min}$ **then**
8: $RLevel_{SD} := active$
9: **end if**
10: $SystemState(j, p) := busy$
11: **end while**
12: **end while**

Algorithm 23 *Pre−scaling at the* $(t+1)th$ *interval*

Require: AResource
1: **while** $JobHandMod_L = busy$ **do**
2: $SystemState(j, p) := busy$
3: **while** $JobHandReslist = JHRL_{active}$ **do**
4: $Predict - NumofSerReq_{t+1} := active$
5: $Calculate - VmUL_{t+1} := active$
6: **if** $VmUL_{t+1} > VMut_{max}$ **then**
7: $Cost - Aware_{P-SU} := active$
8: **end if**
9: $SystemState(j, p) := busy$
10: **end while**
11: **end while**

crease. A maximum utilisation threshold transitions *systemstate* to *busy* as seen in lines 6 to 9.

When more VMs are provisioned with a reciprocal increase in VM utilisation threshold. Self-healing scaling up is activated to ensure that more *Aresources* are provisioned. This supports the increased VM utilisation threshold as seen in lines 10 to 13. Also, the high VM utilisation threshold causes resource-level scaling up to be activated. This utilises unallocated available *Aresources* to scale up the VMs as seen in lines 14 to 16.

However, when the VM utilisation threshold decreases, *Aresources* scaling down is activated as seen in lines 17 to 20. This causes the outputs of job processing to be generated. The simulation duration, average queuing time and average utilisation of PMs are updated to *maximum* values. The *systemstate* is updated to *busy* as seen in lines 21 to 25. This refinement is equivalent to the fourth phase of our ground model as shown in figure 5. The autoscaler by [7] employs three specialised scaling operations in this phase.

Virtual Resource Scaling Down: During job handling, when the VM utilisation threshold is below the expected maximum threshold, VM level scaling down is activated. This causes unused *Aresources* of VMs to be scaled down as seen in lines 3 to 6 of algorithm 22. Also, when the state of the VM utilisation threshold still remains the same, the resource level scaling down is activated as seen in lines 7 to 9. *SystemState* remains updated to *busy* throughout this operation.

Pre-scaling at $(t + 1)^{th}$ **interval:** Also, the number of service requests are predicted during job handling. This activates the computation of VM utilisation threshold at $(t + 1)^{th}$ intervals as seen in lines 3 to 5 of algorithm 23. When the VM utilisation threshold at $(t + 1)^{th}$ interval is greater than the maximum VM utilisation threshold, cost aware scaling up is activated as seen in lines 6 to 8. *SystemState* remains updated to *busy* throughout this operation.

Cost-aware Pre-scaling up: Moreover, during VM level scaling up; when the number of user requests is greater than zero, and the *Aresources* provisioned are not sufficient to handle user requests. The VM with the smallest capacity is activated as seen in lines 3 to 7 of algorithm 24. Conversely, if the *Aresources* are sufficient, a comparison between resource level scaling up or VM level scaling up is made to select the appropriate option as seen in lines 8 to 11. *SystemState* remains updated to *busy* throughout this operation as seen in line 12.

This refinements satisfies our evaluation criteria discussed in sub-section 4.2 since derived functions were applied to evaluate job handling of [7]. Also, the *systemstate* transitioned during the modelling of the phase including the specialised operations which are equivalent to our ground model.

Algorithm 24 $Cost - aware\ \ Pre - scaling\ \ up$

Require: AResource
1: **while** $JobHandMod_L = busy$ **do**
2: $SystemState(j, p) := busy$
3: **while** $JobHandReslist = JHRL_{active}$ **do**
4: $VmLevel_{SU} := active$
5: **if** $NumOfRequests > 0$ **then**
6: **if** $Aresource < NumOfRequests$ **then**
7: $SmallestVM_{SU} := active$
8: **else**
9: $Rlevel_{SU} := active \vee VMlevel_{SU} := active$
10: **end if**
11: **end if**
12: $SystemState(j, p) := busy$
13: **end while**
14: **end while**

5 Conclusion

In this paper, we investigated the issues with evaluating auto-scaling mechanisms. We proposed an ASM model to formalize newly proposed or pre-existing auto-scaling techniques. The development of our ASM model involved meticulous construction, comprising a comprehensive ground model and a set of five ASM Rules. These elements served to capture the fundamental structure of auto-scalers and their essential execution phases. The refinement process employed within our model allowed for thorough scrutiny, ensuring its validity by enabling equivalence checks and formalization of the algorithms. Rigorous validation was carried out using the esteemed CoreASM Simulator, confirming the model's accurate response to various dynamic scenarios and system states.

In future work, we will focus on further fortifying the verification of our model by applying comprehensive temporal properties test cases. This verification process aims to establish the correctness and equivalence of our specifications. To achieve this goal, we will explore and integrate additional model-based testing and verification approaches, harnessing their capabilities to enhance the reliability and robustness of our model.

Acknowledgement

This research was supported by the Hungarian Scientific Research Fund under the grant number OTKA FK 131793.

References

[1] N. Herbst, S. Kounev, and R. Reussner (2013). Elasticity in cloud computing: What it is, and what it is not. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)* pp. 23–27, 2013.

[2] M. A. Netto, C. Cardonha, R. L. Cunha, and M. D. Assunçao (2014). Evaluating auto-scaling strategies for cloud computing environments, In *2014 IEEE 22nd International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE pp. 187–196. https://doi.org/10.1109/mascots.2014.32.

[3] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano (2014). A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing*, vol. 12, no. 4, pp. 559–592. https://doi.org/10.1007/s10723-014-9314-7.

[4] Y. Gurevich (1993) Evolving algebras: an attempt to discover semantics. *Current Trends in Theoretical Computer Science, eds. G. Rozenberg and A. Salomaa, World Scientific.* pp. 266–292 https://doi.org/10.1142/9789812794499_0021

[5] N. Roy, A. Dubey, and A. Gokhale (2011). Efficient autoscaling in the cloud using predictive models for workload forecasting. In *2011 IEEE 4th International Conference on Cloud Computing*, IEEE, pp. 500–507. https://doi.org/10.1109/cloud.2011.42.

[6] C. Qu, R. N. Calheiros, and R. Buyya (2018). Auto-scaling web applications in clouds: A taxonomy and survey. *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–33. https://doi.org/10.1145/3148149.

[7] J. Yang, C. Liu, Y. Shang, Z. Mao, and J. Chen (2013). Workload predicting-based automatic scaling in service clouds. In *2013 IEEE Sixth International Conference on Cloud Computing*, IEEE, pp. 810–815. https://doi.org/10.1109/cloud.2013.146.

[8] E. Börger (2010). The abstract state machines method for high-level system design and analysis. In *Formal Methods: State of the Art and New Directions*, Springer, 2010, pp. 79–116. https://doi.org/10.1007/978-1-84882-736-3_3.

[9] P. Arcaini, R.-M. Holom, and E. Riccobene (2016). Asm-based formal design of an adaptivity component for a cloud system. *Formal Aspects of Computing*, vol. 28, no. 4, pp. 567–595. https://doi.org/10.1007/s00165-016-0371-5.

[10] G. Kecskemeti (2015). Dissect-cf: a simulator to foster energy-aware scheduling in infrastructure clouds. *Simulation Modelling Practice and Theory*, vol. 58, pp. 188–218. https://doi.org/10.1016/j.simpat.2015.05.009.

[11] H. Ghanbari, B. Simmons, M. Litoiu, C. Barna, and G. Iszlai (2012). Optimal autoscaling in a iaas cloud. In *Proceedings of the 9th international conference on Autonomic computing*. ACM, pp. 173–178. https://doi.org/10.1145/2371536.2371567.

[12] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang (2014). Adaptive, model-driven autoscaling for cloud applications. In *11th International Conference on Autonomic Computing ({ICAC} 14)*, pp. 57–64.

[13] M. Dhaini, M. Jaber, A. Fakhereldine, S. Hamdan and R. Haraty (2021). Green computing approaches-A survey. *Informatica*, vol. 45, 2021. https://doi.org/10.31449/inf.v45i1.2998.

[14] D. Saxena and A. K. Singh (2021). A proactive autoscaling and energy-efficient vm allocation framework using online multi-resource neural network for cloud data center. *Neurocomputing*, vol. 426, pp. 248–264. https://doi.org/10.1016/j.neucom.2020.08.076.

[15] A. Al-Dulaimy, J. Taheri, A. Kassler, M. R. H. Farahabady, S. Deng, and A. Zomaya (2020). Multiscaler: A multi-loop auto-scaling approach for cloud-based applications. *IEEE Transactions on Cloud Computing*. https://doi.org/10.1109/tcc.2020.3031676.

[16] Q. Z. Ullah, G. M. Khan, and S. Hassan (2020). Cloud infrastructure estimation and auto-scaling using recurrent cartesian genetic programming-based ann. *IEEE Access*, vol. 8, pp. 17 965–17 985. https://doi.org/10.1109/access.2020.2966678.

[17] A. Belkacem and Z. Houhamdi (2022). Formal approach to data accuracy evaluation, *Informatica*, vol. 46, https://doi.org/10.31449/inf.v46i2.3027.

[18] H. Debbi (2021). Modeling and Performance Analysis of Resource Provisioning in Cloud Computing using Probabilistic Model Checking, *Informatica*, vol. 45, https://doi.org/10.31449/inf.v45i4.3308.

[19] T. LakshmiPriya and R. Parthasarathi, "An asm model for an autonomous network-infrastructure grid," in *International Conference on Networking and Services (ICNS'07)*. IEEE, 2007, pp. 29–29. https://doi.org/10.1109/icns.2007.29.

[20] A. Bianchi, L. Manelli, and S. Pizzutilo (2011), A distributed abstract state machine for grid systems: A preliminary study. In *Proceedings of the Second International Conference on Parallel, Distributed, Grid And Cloud Computing For Engineering, Civil-Comp Press, Ajaccio, France, Paper*, vol. 84. https://doi.org/10.4203/ccp.95.84.

[21] A. Bianchi, L. Manelli, and S. Pizzutilo (2013). An asm-based model for grid job management. *Informatica*, vol. 37, no. 3, 2013.

[22] P. Arcaini, R.-M. Holom, and E. Riccobene (2016), Asm-based formal design of an adaptivity component for a cloud system, *Formal Aspects of Computing*, vol. 28, no. 4, pp. 567–595. https://doi.org/10.1007/s00165-016-0371-5.

[23] E. Börger (2003). The asm refinement method. *Formal aspects of computing*, vol. 15 (2-3): pp. 237–257. https://doi.org/10.1007/s00165-003-0012-7.

[24] J. Fitzgerald and P. Larsen (2009). Modelling systems: practical tools and techniques in software development, Cambridge University Press. https://doi.org/10.1017/cbo9780511626975.