Received 1 August 2024, accepted 12 September 2024, date of publication 25 September 2024, date of current version 10 October 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3467228

RESEARCH ARTICLE

Automated Debugging Mechanisms for Orchestrated Cloud Infrastructures With Active Control and Global Evaluation

JÓZSEF KOVÁCS[1,2], **BENCE LIGETFALVI**[1,3], **AND RÓBERT LOVAS**[1,3]
[1]Institute for Computer Science and Control (SZTAKI), Hungarian Research Network (HUN-REN), 1111 Budapest, Hungary
[2]Centre for Parallel Computing, University of Westminster, W1W 6UW London, U.K.
[3]Institute for Cyber-Physical Systems, John von Neumann Faculty of Informatics, Óbuda University, 1034 Budapest, Hungary

Corresponding author: József Kovács (jozsef.kovacs@sztaki.hun-ren.hu)

This work was supported in part by the National Research, Development and Innovation Office (NKFIH) through OTKA under Grant K 132838; in part by the ÚNKP-22-5 New National Excellence Program of the Ministry for Innovation and Technology from the Source of the National Research, Development and Innovation Fund; and in part by European Union's Horizon Europe Research and Innovation Program under Grant 101131207 (GreenDIGIT). The work of Bence Ligetfalvi was supported by the Doctoral School of Applied Informatics and Applied Mathematics, Óbuda University. The work of Róbert Lovas was supported by the János Bolyai Research Scholarship of Hungarian Academy of Sciences.

ABSTRACT Orchestration methods at Infrastructure-as-a-Service (IaaS) level automate the deployment, scaling, and management of virtualized resources, typically across multiple hosts and data centres. While orchestration provides many advantages, it also introduces several challenges in testing and debugging phases, particularly due to the distributed nature of the virtualized resources. Even the proper initial deployment of interdependent virtual machines (VM) may cause fatal errors since the unpredictable timing conditions may change the overall initialisation method, which can lead to abnormal behaviour, i.e. in complex, non-deterministic environments, the set of VM configurations can drift from their expected states ('configuration drift'). The overall motivation of our research is to improve the reliability of cloud-based infrastructures with minimal user interactions and significantly automate the time-consuming debugging process. This paper focuses on the examination and behaviour of cloud-based infrastructures during their deployment phase. We continued the adaption of a replay-active control based debugging technique, called macrostep, in the field of cloud orchestration. In order to provide efficient support for developers troubleshooting major deployment related errors, the fundamental macrostep mechanisms have been enriched and significantly extended including 1) the automated generation of collective breakpoint sets, 2) parallel and robust traversal method for such consistent global states with 3) automated evaluation of global predicates in each global state of VM set. Furthermore, the novel methods have been 4) generalized towards wider user scenarios by targeting the Terraform orchestration tool as well (besides the already supported Occopus). The paper describes the significantly enhanced approach, our design choices, and also the implementation of the experimental debugger tool with a use case for validation purposes by addressing the deployment of a SLURM (HPC) cluster.

INDEX TERMS Cloud, IaaS, debugging, orchestration, replay, active control, troubleshooting, macrostep, evaluation, global predicates.

I. INTRODUCTION

Modern Infrastructure-as-a-Service (IaaS) cloud computing systems allow automated construction and maintenance of virtual infrastructures [1] leveraging on the concept of virtual machines (VMs) as the fundamental building block and the concept of cloud orchestration [2], [3].

The associate editor coordinating the review of this manuscript and approving it for publication was Ali Kashif Bashir.

A. CHALLENGES

In the fields of distributed computing and cloud computing the following key challenges are discussed.

1) Concerning *complexity and dynamism*, the resource allocation mechanisms may raise issues, since orchestrated systems dynamically allocate resources. Moreover, the software engineers and testers face multi-layer abstraction, i.e. debugging becomes more difficult with multiple layers of virtual machines, containers, and complex orchestration mechanism (see example of MiCADO [4]).
2) Handling the *distributed nature* of the system is unavoidable, multiple nodes and network complexity require complex troubleshooting method in the non-deterministic environment coping with the probe effect, the non-reproducibility and the completeness problem [5]. For instance, let us assume that a given orchestrated cloud deployment scenario always generates correct configuration on a particular cloud platform or on a set of cloud platforms in hybrid and federated clouds (where the software engineers intended to develop and deployed their services) but often fails on other cloud platforms operated by other IaaS providers. Mostly, the reason for this behaviour is the varying relative speeds of deployment tasks together with the untested race conditions in the non-deterministic environments. The different timing conditions might be occurring more frequently on cloud-based platforms than on dedicated clusters or traditional supercomputers because of the different implementation of the underlying operating systems/communication layers and the unpredictable network traffic, CPU loads or other dynamical changes in the multi-tenant environment. The above-described phenomenon can be very crucial because one cannot ensure that the cloud-based deployment always provisions (captures) the same computing nodes with almost the same timing conditions in case of (re)deployment or VM failure.
3) The testers have to tackle with *configuration and dependency* related problems, including the phenomenon of 'configuration drift' when the configurations for a set of often interdependent VMs can drift from their intended states, leading to software bugs, e.g. failed deployments. For detection of such drifts in complex or large scale scenarios, *cloud logging and monitoring tools* [6] may aggregate logs that become too large to manage effectively. Furthermore, metrics overload may occur, i.e. too much data can obscure the real issues in case of failed deployment.
4) Most of the existing *debugging tools with limited capabilities* may not scale or adapt well to orchestrated IaaS environments (see Section II).

Furthermore, *specialized knowledge* is also often required from testers, raising a barrier to effective debugging or troubleshooting. *Security restrictions* can also limit debugging capabilities, and debugging itself might expose sensitive data, complicating the entire process. These last two topics are out of the scope of this paper.

B. MOTIVATION AND GOALS

Reference architectures serve as foundational blueprints in cloud computing, encapsulating best practices and facilitating standardization across deployments. Their reliability is crucial for several reasons. First, they guide the implementation by adequately designed and tested components together with proven technologies, thereby reducing potential points of failure. Second, they ensure interoperability and consistency, essential for system-wide reliability. Additionally, these architectures help in risk mitigation by offering guidance and extensive documentation on failure scenarios. Furthermore, they provide a scale-able and future-proof framework, crucial for maintaining reliability as the system evolves. Given their role in shaping cloud systems, the reliability of reference architectures is not merely an added benefit, but a critical requirement for ensuring the overall credibility of cloud computing solutions.

Our research motivation is three-fold: (i) automate further the mechanisms for the development of reliable cloud-based research infrastructures leveraging on the concept of reference architectures, (ii) accelerate the debugging process taking the benefits of feasible parallel testing of the given reference architecture candidate, (iii) significantly reducing human interactions during the automatised and accelerated debugging process of reference architecture candidates. In this way, the development and operation costs of research infrastructures might be significantly reduced, and the cloud users may get a higher quality of service. This paper focuses on all these motivations.

C. BACKGROUND AND PROPOSED APPROACH

According to the literature [5], the *distributed debugging methodologies* can be categorised according to the level of support they provide to the software developers and testers concerning the global predicate specification and detection,

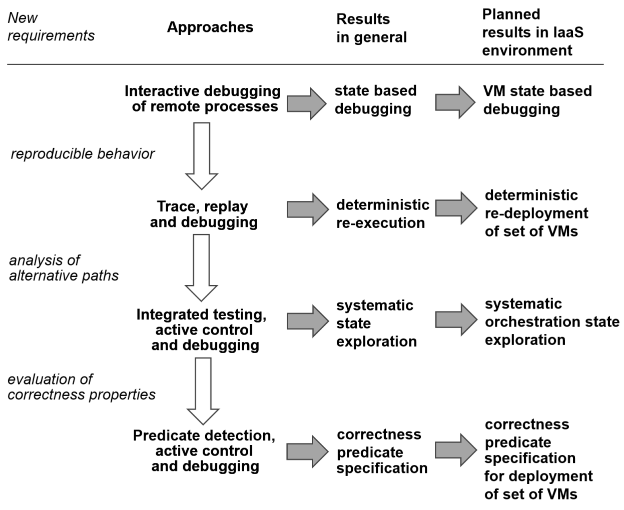

New requirements
Approaches
Results in general
Planned results in IaaS environment
Interactive debugging of remote processes
state based debugging
VM state based debugging
reproducible behavior
Trace, replay and debugging
deterministic re-execution
deterministic re-deployment of set of VMs
analysis of alternative paths
Integrated testing, active control and debugging
systematic state exploration
systematic orchestration state exploration
evaluation of correctness properties
Predicate detection, active control and debugging
correctness predicate specification
correctness predicate specification for deployment of set of VMs

FIGURE 1. Classification of general distributed debugging methodologies [5] extended with our planned results in orchestrated IaaS environment.

and the search for the origin/cause of the special software bugs in the distributed program.

In this section, these approaches are briefly introduced based on [5]. The outlined methods begin from the most basic one, and the introduced four approaches are complementary to each other, i.e. each of them attempts to extend and also eliminate the barriers of the previous ones (see Figure 1).

In case of remote sequential processes, *interactive debugging* is heavily relying on (or a sort of extension of) the widespread sequential software debugging functionalities. This approach enables the online observation as well as the control of the execution of sequential and remotely running processes one-by-one. This method can be used for examining only the local states and histories of individual processes; it is considered as its main drawback.

In order to address the complex issue of *non-reproducibility*, the 'Trace, replay and debugging' approach collects traces (see 2nd row in Figure 1). The trace contains of a set of well-selected (relevant) events generated and collected during the first execution of the distributed program. The collected trace represents a so-called computation path (i.e. a consistent run) that can be analysed post-mortem (after the execution); if one or more suspicious or erroneous situations are found, then the software developer or tester can run the distributed program again but under the strict control of a proper supervisory mechanism. During the re-execution, the traced sequence of events is used to force the distributed computation to follow and repeat exactly the same computation path with the same timing conditions.

The *integrated testing, active control and debugging* attempts to tackle the barrier of the simple and passive log collection and replay approach (see 3rd row in Figure 1). The proposed solutions are different ones concerning the method they produce the desired computation path (i.e. consistent run), which is to be followed during a controlled execution.

At the most advanced level, the main goal of the *predicate detection, active control and debugging* approach is to help the user increasing the confidence on the outcome of the previous approach and highly automate the verification (see 4th row in Figure 1). That is why, it allows the user to specify the correctness criteria using so-called global predicates. These global predicates are automatically evaluated by special detection algorithms during the consistent runs.

One feasible way to prove the reliability (including the cloud platform-agnostic feature) of complex deployment and maintenance strategies is to leverage on advanced systematic debugging/troubleshooting methods in order to find the timing or architecture dependent failures in the designed deployment description and orchestration steps. For this purpose, our research team applied and extended the *macrostep-based debugging* [7] methodology that has been introduced originally for message-passing parallel programs and developed in the P-GRADE graphical programming environment [8]. The macrostep-based debugging not only follows the "Automated detection of global predicates, active control and debugging" approach but also extends it with some useful mechanism and functionalities.

The main essence of the *macrostep methodology* is to discover the different timing combinations among the concurrent behaviour of distributed events. While in the message-passing environment, the communication primitives were analysed by macrostep, in cloud orchestration, the parallel deployment of entities (software components on virtual machines) represent the source of possible concurrent behaviour. In the original macrostep technology, the central controller was able to enforce the program to be executed in a way to simulate or enforce all possible timing conditions, i.e., to execute the program through all possible execution paths to find the issues related to concurrent behaviour. The technique is based on applying (collective) breakpoints at certain points in the execution of concurrent activities performed by the various software components. Once the breakpoints are set, the execution of the components are controlled in a way to traverse the entire deployment process towards extreme, unexpected situations that may have a low chance in normal conditions.

For *orchestration*, the goal is to apply this technique to investigate the behaviour of the VM deployment steps of a cloud orchestrator for a particular cloud reference architecture, i.e., a network of VMs with installed software components and services interconnected to each other as they described in the last column ('Planned results in IaaS environment') in Figure 1. As the result, not only the deterministic re-deployment of a set of VMs but the systematic exploration of the state space of orchestration process becomes available. Moreover, the correctness properties (e.g. the undesired 'configuration drifts') in the orchestrated deployment process of a set of VMs might be detected as well by properly defined global predicates.

The first experimental prototype of our macrostep debugger for cloud was introduced in paper [9] based on the Occopus cloud orchestrator [10], [11] framework, and supported debugging at the level of "Integrated testing, active control and debugging" (see 3rd row in Figure 1). Occopus is an open-source cloud orchestrator software tool that has been designed and implemented by SZTAKI to support the research and experiments in the field of cloud orchestration. Similarly to other orchestration tools, Occopus applies the de-facto standard Cloud-init tool to contextualise the newly launched virtual machines in order to start the services. The contextualisation itself is the place where the deployment of the infrastructure can be manipulated, i.e., artificially influenced in order to reach the desired (faulty) behaviour of the reference architecture.

In order to provide efficient support for developers troubleshooting major deployment related errors, the first experiment prototype has been enriched and significantly extended, including the following methods:

1) the automated generation of collective breakpoint sets at the 'Trace, reply, and debugging' level,

2) the parallel and robust traversal method for such consistent global states at 'Integrated testing, active control and debugging' level, together with
3) the automated evaluation of global predicates in each global state of VM set at the 'Predicate detection, active control and debugging' level.

Moreover, these novel methods have been generalized towards wider user scenarios by targeting the Terraform orchestration tool as well (besides the already supported Occopus). The paper describes the significantly enhanced approach, our design choices, and also the implementation of the experimental debugger tool with a use case for validation purposes by covering the whole orchestrated deployment process of a SLURM [12] (HPC) cluster.

The rest of paper is structured as follows:

- Section II introduces related works on the field of debugging and troubleshooting on cloud orchestration,
- Section III helps understanding the concept of Macrostep-based debugging and collective breakpoint-based troubleshooting,
- Section IV describes the architecture and operation of the new version of the prototype, and dives into the descriptions of the interdependent components and operation mechanisms,
- Section V is about the abstraction of orchestration and application scenarios (towards other orchestrator tools such as Terraform),
- Section VI provides details on the new automated breakpoint generation methods for both supported orchestrator tools,
- Section VII introduces the new facilities for the automated evaluation of global predicates describing/evaluating the expected behaviour (correctness) of the orchestrated deployment process of VMs,
- Section VIII shifts the focus further for real life application scenarios by introducing the parallel traversal of paths in the given execution tree (state space) for accelerating the debugging and testing process,
- Section IX contains the validation of the recently elaborated mechanisms and debugger prototype with a use case (SLURM cluster deployment),
- Section X contains conclusions and future works.

II. RELATED WORK

In the literature, some related research achievements are described for e.g. HPC applications [13], but the typical example is the remote cloud debugging [14] feature for Windows Azure Cloud Services [15] that can be considered the most basic functionality of all distributed debuggers. Oczan et al. [16] extends the remote debugging towards containerized applications in edge computing environments.

Smara et al. [17] proposed a fault detection method for clouds leveraging on the concept of acceptance tests. In their framework, the main aim is to construct fail-silent cloud components, which have the ability of self-fault detection. On the contrary, our approach can create a series of consistent global cuts (or states) for error checking and debugging in the distributed environments.

According to Zhang et al. [18] two main categories of cloud fault detection can be distinguished: rule-based or statistical detection. Rule-based detection methods are built on simple rule sets on the error message and record components, or basic decision trees can be built using multiple rules and queries. Our approach is to support both categories. However, our work focuses on the mechanism of how to traverse the state space where such rule-based or statistical approaches might be applied later on.

Goossens et al. [19] also introduced a communication-centric debugging method, but their work covers the problem at the hardware level and their work is oriented towards transactions.

Debugger [20] developed by Google is able to follow the application states in real-time without the need of slowing down or stopping the application. It can take a snapshot from a desired state of the program, i.e., the call stack, and the variable values are searchable and verifiable. Another important feature is the so-called logpoints, which is able to create custom string and variable logging mechanisms in the code. This debugger cannot provide (among others) advanced functionalities for handling multiple, orchestrated VMs.

Merino and Otero [21] enhanced a record and reply mechanism for microservice debugging by a checkpoint and restart mechanism for software containers. The proposed architecture and solution can handle the non-deterministic execution environment but limited to microservices, and active control and testing features are not supported.

Quroush and Ovatman [22] described a record and replay mechanism for cloud-based multi-tenant services that enables software developers to debug their application in the replay phase after a failure detected in the recording phase. Their approach has promising results, but there is no support for systematic traversing of the state space and is limited to script-based deployments.

Sharma et al. [23] developed an endpoint-based monitoring solution that allows tenant-level monitoring. In addition, the system stores the history of the metric data for other uses, e.g., history evaluation or validation for SLA compliance. The solution offers monitoring customisation, extensibility and portability, but the use of the tool is limited to OpenStack-based cloud resources, and there is no active control mechanism offered for debugging purposes.

Gan et al. [24] have developed an online cloud performance debugging system (Seer) leveraging on big data to navigate the complexity of cloud debugging. Moreover, deep learning models had been used for proactive QoS violation detection of cloud microservices. Seer applies distributed tracing and copes with the unpredictable timing conditions similarly to us but their main scope is the performance (and not correctness) debugging of the interdependent components.

Baek et al. [25] created a monitoring solution where special loggers are inserted into the cloud components. The solution processes the logs and creates a resource graph from them.

The resource graph presents the changes of the resource events and can be queried to define previous state changes. However, the functionality of the tool is limited because it does not investigate the guest system in detail and needs to inject the components into the system. It works only in an OpenStack environment without active control or other advanced debugging mechanisms.

Cotroneo et al. [26] described a run-time failure detection method via non-intrusive event analysis in a large-scale cloud computing platform. They follow a black-box tracing approach, and able to perform run-time verification (checking correctness of a system execution on-the-fly according to specific properties) while executing a campaign of fault injection experiments in a multi-tenant scenario using OpenStack. Despite several similarities, our active control attempts to provide a more generic solution focusing on the non-deterministic behavior of the examined cloud system, and leveraging on white-box approach.

Some promising methods are described by Manner et al. [27] to troubleshoot serverless functions in the cloud with a combined monitoring and debugging approach. In their paper, a semi-automated troubleshooting process is presented to improve fault detection and resolution based on enhanced log data quality, automatically detected failed executions with test skeletons. Similarly to our approach, their method leads to an increased test coverage, a better regression testing and more robust functions of cloud-based systems but they narrowed the scope to one particular class of cloud applications with serverless functions.

Gulzar et al. introduced not only new debugging primitives in their BigDebug [29] tool (e.g. simulated breakpoints, guarded watchpoints and forwards/backward tracing) for interactive big data processing in Apache Spark but automated debugging of big data analytics in data-intensive scale-able computing as well. BigSift [28] adds even more advanced testing features to their solution by so-called Delta Debugging. One of the main limitations of their proposed solution is the tight binding to the Apache Spark framework.

Our team is involved in a joint research where we have already published some of our experiments and promising results and experiences with deep learning enhanced steering mechanisms for debugging of fundamental cloud services [30]. The main objective of our work was to start integrating some selected stochastic modelling and verification techniques based on deep learning methods into the debugging cycle in order to handle large state spaces more efficiently, i.e. by steering the process of traversing state space towards suspicious situations that may result in potential bugs in the actual system.

Table 1 contains an overview of the ten most relevant implementations discussed in this section. As can be seen in the table, all of these provide basic functionalities like tracing (*trace & replay* column), but only half of them have replay functionalities, making them capable of reproducing previous runs. While half of these implementations provide correctness checking, most of them do not utilize active control or integrated testing. Only three out the ten implementations in the table provide full-scale debugging functionalities, starting from tracing to correctness checking. In summary, implementations with tracing capabilities are present in a variety fields (cloud, edge, microservices), while more complex debugging systems are limited to certain parts of cloud computing.

By combining the macrostep debugging (see Section III) methodology and the features of Occopus and Terraform cloud orchestrators, the presented work attempts to overcome the limitation of existing debugging solutions since not even the most widely used cloud providers, nor the state-of-the-art debugger tools offer high level and advanced debugging facilities to their users that are similar to the described macrostep-based concept.

III. MACROSTEP-BASED DEBUGGING

A. ORIGINAL CONCEPT FOR MESSAGE-PASSING PROGRAMS

During debugging parallel and distributed programs, several aspects have to be taken into account, one being that parallel programs show non-deterministic behaviour, making the reproduction of erroneous runs a difficult problem (see details in Section I). This can be due to several factors, like differing relative CPU and/or memory speeds, operating system scheduling, network latency, etc. Another aspect is that sequential debugging methods, like breakpoint-to-breakpoint execution, cannot be applied to parallel programs only with strong limitations.

Early debugging methods in parallel systems relied on the so-called "monitor & replay" approach [7]. In the first monitoring (or recording) phase, the monitoring tool collects as much data about the parallel program as required to reproduce the run in a deterministic way in the second replay phase. In this approach, replay is driven by the previously gathered program information. This approach does provide a solution for the debugging of parallel programs, but a new problem, the probe-effect is introduced. This means that the monitoring of the parallel program affects its timings. Although one can mitigate the impact of the probe-effect by reducing the amount of collected information during monitoring, the effect cannot be completely eliminated itself.

Another approach for parallel program debugging is the so-called "control & replay" method (or active control), in which we make use of systematically generated test cases to exhaustively and completely test every possible timing condition in the parallel program. Replay is not performed according to collected data, but according to the generated test cases. The most important part of this debugging approach is to find a suitable solution that can generate these test cases [7], [31].

The aforementioned macrostep-based debugging methodology utilises this "control & replay" approach by introducing the concept of collective breakpoints and macrosteps. An early implementation of this debugging method was DIWIDE (DIstributed WIndows DEbugger) [7].

TABLE 1. Comparison of the ten most relevant debugging methods which have an implementation.

Reference	Main application field	Trace & Replay	Active control & Integrated testing	Correctness evaluation
Snapshot Debugger [20]	PaaS, cloud applications	tracing (via snapshots)		yes
Merino et al. [21]	microservices	yes (monitor & replay)		
Quroush et al. [22]	multi-tenant services, cloud applications	yes (monitor & replay)		
Sharma et al. [23]	OpenStack-based clouds	tracing		
Gan et al. [24]	microservices, QoS violation detection	tracing	yes (via deep learning)	
Baek et al. [25]	OpenStack-based clouds	tracing		
Cotroneo et al. [26]	OpenStack-based clouds	tracing		yes
Manner et al. [27]	FaaS, serverless functions	yes	yes	yes
Gulzar et al. [28]	Apache Spark framework	yes	yes	yes
Lovas et al.	IaaS, cloud orchestration	yes (control & replay)	yes	yes

Macrostep-based debugging builds on the following concepts: local breakpoints, collective breakpoints, macrosteps, the execution tree and meta-breakpoints.

Local breakpoints are implemented on the process level, and a process is halted when it hit a local breakpoint. A collective breakpoint is a set of local breakpoints, preferably covering each process. If a collective breakpoint contains local breakpoints from every process, then it is a complete one, otherwise it must be considered partial. If a collective breakpoint contains local breakpoints for all possible alternative paths for each process, then the collective breakpoint is strongly complete. A macrostep is the executed code region between two collective breakpoints. The original macrostep-debugging concept distinguishes pure and compound macrosteps, meaning that if communication-related code is found only as its last element, then the macrostep is pure, otherwise it is compound. By using strongly complete, pure collective breakpoints, the traditional breakpoint-to-breakpoint debugging methodology of sequential programs can be extended to parallel programs. While in sequential programs debugging is done in a step-by-step manner, in parallel programs, it can be achieved macrostep-by-macrostep [7].

One order of consecutive collective breakpoint hits is an execution path. The execution tree contains all execution paths, all possible timing conditions in the parallel program. It is built up of collective breakpoints and macrosteps, collective breakpoints being the nodes and macrosteps being the directed edges. The execution tree starts with the root node. In the original macrostep concept there are 3 distinct type of nodes: fork, alternative and deterministic. Deterministic nodes do not create new execution paths, however at alternative and fork nodes, it is possible to force the parallel program towards different execution paths. Breakpoints can be placed in the execution tree as well, in which case they are called meta-breakpoints, essentially meaning that the parallel program is steered to a specified node in the execution tree. Using an appropriate debugging tool that can generate suitable collective breakpoints [7], one can achieve the complete [31] and exhaustive testing [32] of parallel programs.

B. MACROSTEP-BASED DEBUGGING IN CLOUD ORCHESTRATION

IaaS systems can contain up to dozens or even thousands of VMs, and contextualisation is usually done in parallel to speed up deployment. It might be a challenging task to locate and analyse errors due to the inherently non-deterministic nature of cloud resources. Contextualisation processes may depend on each other, VMs differing in configuration, the actual load on physical resources used by the infrastructure (physical CPU, memory, storage, etc.), among others, can all influence timings during the deployment of a given infrastructure (or reference architecture). Because of this non-deterministic behaviour, it is not unusual that erroneous situations cannot be reproduced reliably.

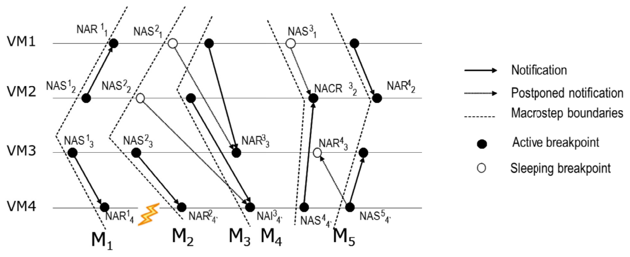


FIGURE 2. Macrostep concept in IaaS cloud infrastructures.

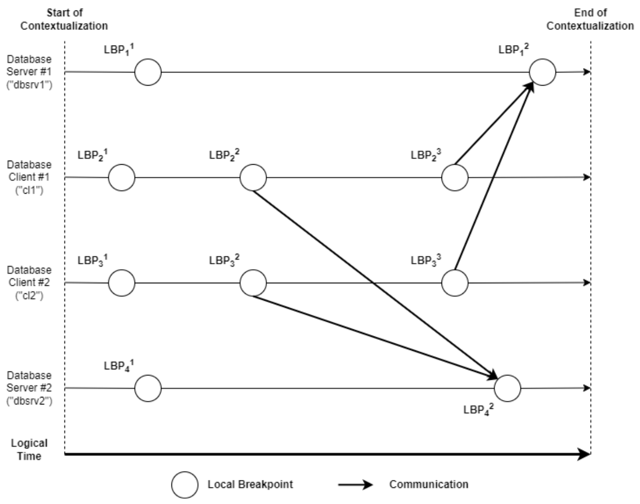

Start of Contextualization
End of Contextualization
Database Server #1 ("dbsrv1")
Database Client #1 ("cl1")
Database Client #2 ("cl2")
Database Server #2 ("dbsrv2")
Logical Time
LBP1 1
LBP1 2
LBP2 1
LBP2 2
LBP2 3
LBP3 1
LBP3 2
LBP3 3
LBP4 1
LBP4 2
Local Breakpoint
Communication

FIGURE 3. An example infrastructure containing four virtual machines.

Traditional parallel systems and infrastructure deployments show many similarities, like potentially dependent processes running concurrently, errors related to wrong timing and problematic error reproduction. Thus, it seems promising to apply the original macrostep debugging methodology (see

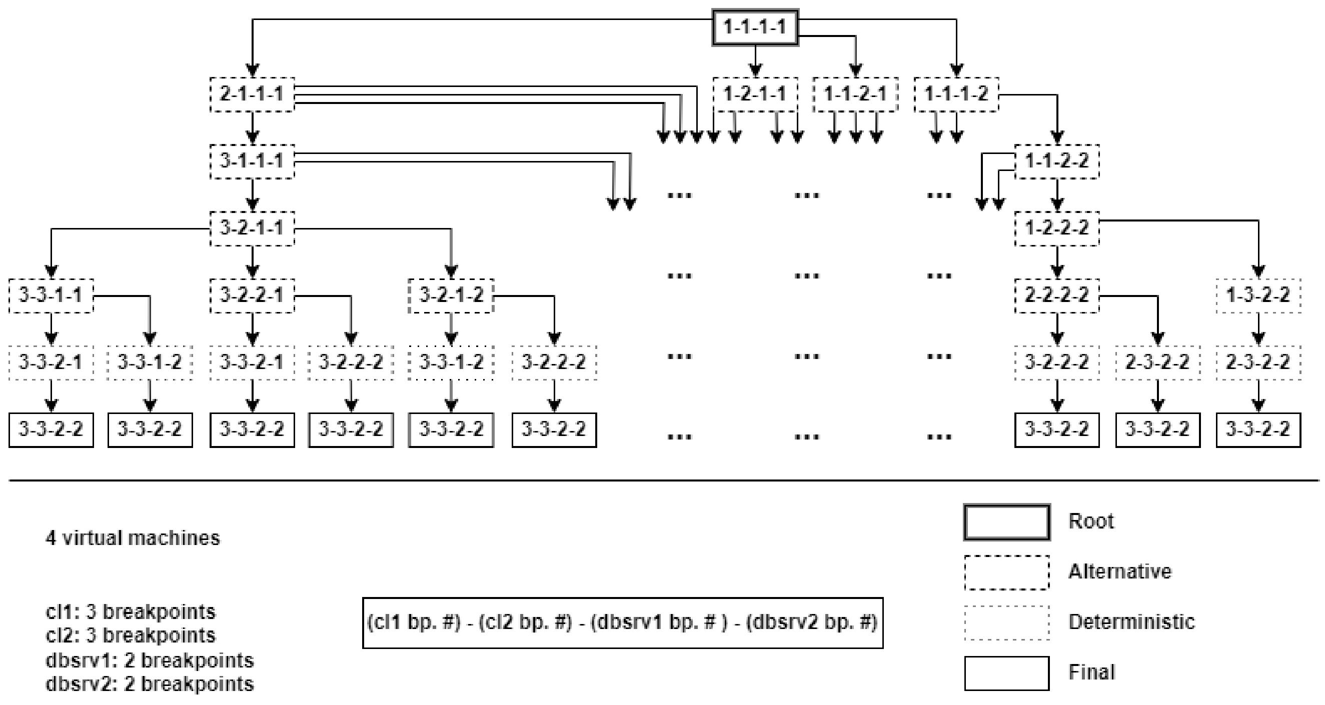


FIGURE 4. Complete execution tree of the example infrastructure with four VMs and ten local breakpoints.

Figure 2) to cloud infrastructure deployment. Similarly to the original concept, processes are running concurrently, which, in the case of IaaS systems, are the contextualisation processes of the VMs. By placing local breakpoints in each virtual machine's contextualisation process (see VM1 to VM4 in Figure 2), the debugger can effectively suspend them until an appropriate control signal is received. Then local breakpoints can be organised into collective breakpoints, each collective breakpoint (see M1 to M5 in Figure 2) containing one local breakpoint from every contextualisation process.

Reaching a collective breakpoint in this context essentially means the overall deployment of the infrastructure is temporarily halted, forming a consistent global cut (or state). At this point, each contextualisation process is waiting to proceed. If the debugger may choose from multiple contextualisation processes to continue, then the collective breakpoint is an alternative collective breakpoint. The next collective breakpoint can be reached by permitting one of the waiting processes to progress. If there is only one process waiting, then the collective breakpoint is deterministic.

The execution tree contains all execution paths, all possible timings that can occur during an infrastructure's deployment. The execution tree's root node is the set of the first local breakpoints from each contextualisation process and is the first collective breakpoint for all execution paths. Nodes in the execution tree are connected by macrosteps, meaning that a macrostep is the executed contextualisation code between two collective breakpoints. In this way, IaaS system deployment can be carried out macrostep-by-macrostep, going from one collective breakpoint to another.

C. EXAMPLE FOR ORCHESTRATED INFRASTRUCTURE DEPLOYMENT

An example (see Figure 3) for demonstrating the benefits of macrostep-based debugging could be an infrastructure containing two database-clients ("cl1" and "cl2") and two database-server ("dbsrv1") and ("dbsrv2").

Normally, timings can be correct in one deployment, and clients would be able to connect to the already existing database, but in another deployment it is possible that the database-server is far behind in contextualisation to handle any read-write request coming from the clients. With the macrostep debugging method, the debugger can systematically test each timing condition and find sufficient ones where clients are able to connect to an already existing database.

A synthetic execution tree has been generated manually for illustration purposes in Figure 4 to understand the nature of an execution tree using this example infrastructure. It has the collection of all possible execution paths combined into a tree at the point of branches. In this tree, the infrastructure has two clients and two servers. The clients contain three breakpoints, while the servers contain two breakpoints. Each node represents a collective breakpoint and a 4-tuple in the rectangles shows which breakpoints the virtual machines are blocked in the order of client1, client2, server1 and server2. On the top, marked by a rectangle with thick line, is considered as the root node. That is the initial point of every virtual machine waiting on their first breakpoints ("1-1-1-1"), i.e., on the first collective breakpoint, which can also be seen in Figure 2 denoted by M1. The final status

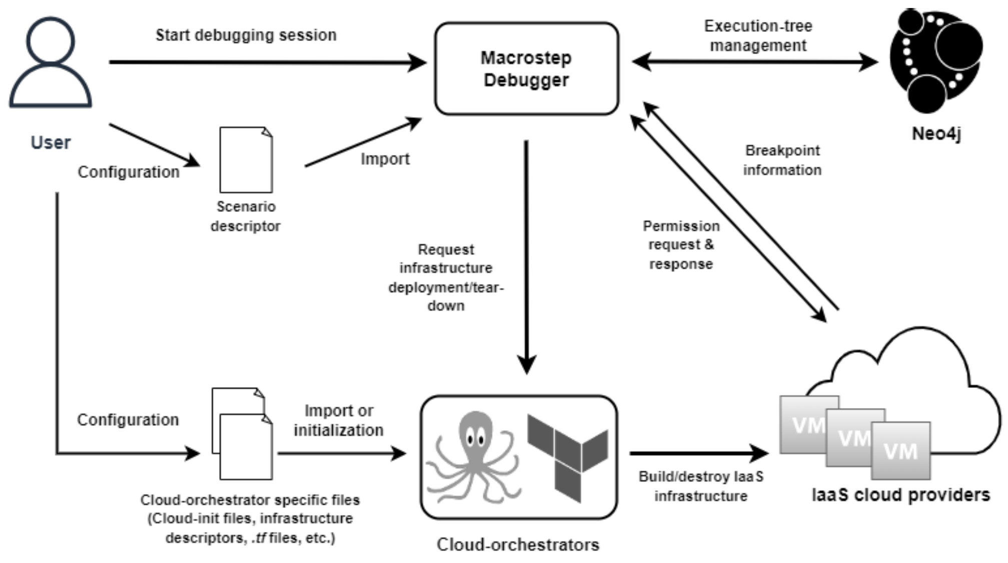

Start debugging session
Macrostep Debugger
Execution-tree management
Neo4j
User
Configuration
Scenario descriptor
Import
Breakpoint information
Permission request & response
Request infrastructure deployment/tear-down
Configuration
Import or initialization
Cloud-orchestrator specific files (Cloud-init files, infrastructure descriptors, .tf files, etc.)
Cloud-orchestrators
Build/destroy IaaS infrastructure
VM
VM
VM
IaaS cloud providers

FIGURE 5. High-level architecture of the experimental integrated debugger system.

for the entire infrastructure arrives when they are blocked on their final breakpoints, which is marked with the label of "3-3-2-2" on each leaf drawn by thin line rectangle back in Figure 4. Between the root and final collective breakpoints, we considered two more types of collective breakpoints in this example. They are the deterministic collective breakpoints marked by the dotted line and the alternative collective breakpoints marked by dashed line. The former means that there is only one path leading to the next collective breakpoint, while the latter represents a branch, i.e., more than one paths leads out from that node to the next ones.

Looking at this execution tree, one may recognise that several nodes in the tree are identical regarding their status, i.e., these collective breakpoints represent the same consistent global status. For example, in Figure 4, the node "3-2-2-2" can be found more than once since these states can be reached through different execution paths. The most extreme example is the collective breakpoint is the one denoted by "3-3-2-2", which is the final one since every path leads to this collective breakpoint representing the final state. By merging the common nodes in this tree, a new way of visualisation of this graph could be created, where the graph starts with one root node and ends with one final node. This conversion is now skipped in this visualisation and the tree is kept just for the sake of understand-ability and for the sake of not losing information on which path a given node has been reached.

Using an appropriate tool, the macrostep-based debugging of orchestrated cloud infrastructures can be achieved. With this method, developers can systematically and exhaustively test every possible timing condition that can arise during the infrastructure's deployment.

IV. PROTOTYPE: MACROSTEP DEBUGGER FOR CLOUD-ORCHESTRATION

In order to prove the viability of our approach, we have developed a prototype for debugging cloud reference architectures. The prototype utilises Occopus [11] and Terraform [33] to orchestrate cloud resources, Cloud-init [34] for the contextualisation of virtual machines, Neo4j [35] graph database for storing and visualising the execution tree, and a central Macrostep Debugger to coordinate the operation of the supporting components. Figure 5 shows the components and their interactions.

A. COMPONENTS

This subsection aims to introduce the components of our prototype one after the other. The next subsection will detail how they cooperate to realise macrostep debugging.

Occopus (see Occopus icon inside Cloud-orchestrators box in Figure 5) developed by our laboratory is a lightweight cloud-orchestrating tool that supports a wide variety of cloud providers, including public (e.g. Amazon Web Services [36], Azure [15]), private, community and hybrid ones as well. It supports infrastructure management during its full life cycle, including deployment, monitoring, scaling and shutdown [10].

To create an infrastructure, Occopus uses a so-called infrastructure descriptor that describes the infrastructure components at a higher level as well, as the dependencies among them. Infrastructure descriptors contain additional information, such as scaling parameters and user-defined variables. While the infrastructure descriptors describe what to create, a node definition (one level down) describes how to create the nodes. An infrastructure descriptor may refer to several node definitions specifying the various features of the nodes (virtual machines). These features are image ID, flavour, network settings and many more cloud-related settings, as well as the way how contextualisation should happen. Contextualisation is realised by Cloud-init to perform changes on the newly created virtual machine. Furthermore, Occopus is able to utilise configuration manager

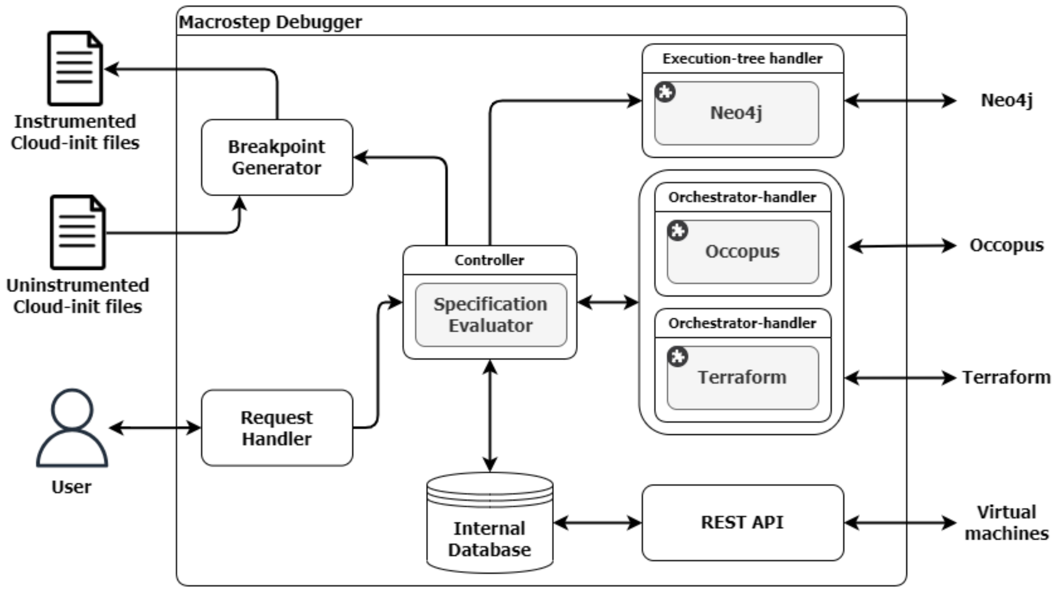


FIGURE 6. Internal architecture of the cloud-based macrostep debugger.

tools like Chef [37] and Puppet [38], and it supports various health-checking procedures (e.g. ping, port, URL, etc.).

Occopus comes with a built-in REST API that makes it easier to build, maintain, scale and destroy infrastructures or nodes remotely. Alternatively, Occopus itself can be used as a CLI tool if needed. Upon the creation of an infrastructure, Occopus allocates unique identifiers both for the infrastructure and for the nodes comprising it. In our initial version of Macrostep Debugger, Occopus was integrated since it is open-source, easy to configure and use, has a rich feature set and is compatible with most cloud platforms.

Terraform (see Terraform icon inside Cloud-orchestrators box in Figure 5) is a cloud orchestration tool [33], which is developed by HashiCorp, and it provides similar functionalities to Occopus. Terraform is capable of deploying, maintaining, scaling and tearing down cloud infrastructures. It can utilize private, public, community and hybrid cloud providers. It is primarily working from so-called *.tf* and *.tfvars* files. These files contain resource definitions, variables, templates (for example for Cloud-init files) that will be used for infrastructure deployment and maintenance. Resources are essentially different components of the infrastructure, for example virtual machines. The aforementioned *.tf* and *.tfvars* files are stored in a working directory that in Terraform terminology is called a root module. Compared to Occopus, Terraform does not include a REST API, making remote access and remote usage slightly more difficult.

Since Terraform is a mature and widely used cloud orchestration tool, capable of deploying and managing highly complex infrastructures, integration with it enables the Macrostep Debugger to provide debugging functionalities for infrastructures deployed through Terraform.

Cloud-init (see bottom left corner in Figure 5) is a de-facto industry standard tool for VM contextualisation, supported by a large set of cloud providers and major Unix distributions. It enables infrastructure developers to customise VMs according to their requirements. It supports handling of users, files, permissions, network settings, configuration managers, packages, disks and partitions. A Cloud-init script can contain different sections, of which one is *runcmd*, that can be used to run user-defined, operating system specific code. Files with custom content can be added to the *write_files* section as well [34]. We will rely on its features when realising the Macrostep technique.

Neo4j (see upper right corner in Figure 5) is a robust, powerful graph database platform, suitable to store, handle and visualise large graphs (e.g. execution trees for Macrostep). Since the number of local breakpoints and execution paths may increase dramatically in certain situations during Macrostep execution, it was necessary to select a solution that could handle a large set of nodes, links and their related information.

The Macrostep Debugger (see Figure 5) is the core component of the system, responsible for the coordination of debugging sessions, like performing manual debugging, automatic debugging through active control, replaying or free-runs. The debugger instructs the orchestrators (Occopus or Terraform) to deploy or destroy the entire infrastructure, decides when to continue the deployment of a particular virtual machine, and it is responsible for managing the execution tree by instructing the graph database. The debugger provides a command line user interface to receive commands. The Macrostep Debugger contains functionalities for evaluating specifications at collective breakpoints, and it is capable of generating the local breakpoints necessary for the Macrostep technique.

One of the most essential parts of the prototype is the Macrostep Debugger component, which is shown with its internals and interfaces in Figure 6. The central part of the Macrostep Debugger is the *Controller* which is responsible for decision-making, initiating communication with other components (orchestrators, user, graph database, virtual machines) and in general, keeping the entire system in operation. It instructs and controls all the other components. The *Request handler* provides an interface for incoming user commands, e.g., for importing scenario descriptors, generating local breakpoints, continuing execution, starting debugging sessions, etc. All these incoming requests are translated into commands for the Controller.

The Controller also instructs the *Orchestrator handler* to create or destroy infrastructure instances. The Macrostep Debugger utilizes a *REST API*, which responsible for communicating with the virtual machines, i.e., receiving notifications on local breakpoints being hit and instructing the virtual machines to continue their deployment. The Macrostep Debugger includes an *Internal database* (implemented via SQLite [39]) for storing the internal status of each debugging session. The *Execution tree handler* performs execution tree related operations, such as the creation or update of nodes in the execution tree. The *Breakpoint generator* is responsible for analysing Cloud-init files and then inserting necessary local breakpoints, creating instrumented contextualisation files.

The Macrostep Debugger requires an infrastructure-independent configuration file called *Scenario descriptor* containing the necessary information of the external components, e.g., endpoint of the Neo4j graph database, type and endpoint of the cloud orchestrator, as well as the name of the infrastructure, etc. The Scenario descriptor also contains a list of specifications that will be evaluated during debugging. This descriptor is loaded by the debugger into the *Internal database*. Further details about the Scenario descriptors can be found in Section V.

B. OPERATION

In order to execute and debug the deployment of an infrastructure, a complete set of descriptors (infrastructure descriptor, node definition and Cloud-init file) is needed for the Occopus cloud orchestrator. Terraform requires resource, variable and template definitions. We assume that Occopus and Terraform are both installed and properly configured, and the necessary cloud credentials are provided to access the cloud API through which the infrastructure is about to be deployed. Once a ready-to-use infrastructure was imported (in case of Occopus) or initialised (in case of Terraform), the orchestrators are ready to use. The user has to import a Scenario descriptor into the Macrostep Debugger after which macrostep debugging can be started.

The *Execution tree handler* (Neo4j plugin) and the Orchestrator handlers (Occopus and Terraform plugins) use the information provided in the Scenario descriptors to find their external tools. Since the contextualisation of the virtual machines is implemented by Cloud-init files, they have to be instrumented to realise breakpoints. If needed, users can utilize the breakpoint generation functionality of the debugger. The Breakpoint generator analyses Cloud-init files and inserts local breakpoints automatically. Further information on breakpoint generation can be found in Section VI.

With *local breakpoints* inserted into the Cloud-init files, the debug session can be started by launching the Macrostep Debugger with the infrastructure name and a type of debugging mode (e.g.: automatic, manual debugging, replay, free-run). First, the Macrostep Debugger instructs Occopus or Terraform to create an infrastructure instance, after which the debugger receives or retrieves the unique ID of the infrastructure instance.

The *cloud orchestrator* starts building the infrastructure and the virtual machines begin their contextualisation according to the Cloud-init files. When a virtual machine hits a local breakpoint during its contextualisation, a breakpoint script is executed, suspending contextualisation until permission for continuation is granted by the debugger.

The macrostep local breakpoint is realised by a script deployed by the *write_files* section in the Cloud-init files, so its invocation can be inserted at the boundary of service installations. This special shell script (called *breakpoint script*) temporarily suspends contextualisation, collects information about the virtual machine's inner state, sends it to the breakpoint register and periodically polls for permission to continue contextualisation. Arguments can be passed to the breakpoint script to better describe operations previously executed during contextualisation. The last call is expected to contain the argument 'last' or 'last_bp' to indicate for the Macrostep Debugger that no more breakpoint exists for the virtual machine and that its deployment is finished. The breakpoint script creates a JSON [40] structure containing the necessary information. The unique ID of the virtual machine along with the name of the infrastructure instance and the virtual machine are part of the JSON descriptor by default for identification. The JSON descriptor can also be extended with user-defined key-value pairs as well. When the script is invoked, it establishes a connection with the REST API of the debugger and posts the JSON structure.

At startup, the Macrostep Debugger waits until the infrastructure reaches *root state* which means that every virtual machine has reached its first breakpoint. This is considered as the root collective breakpoint, i.e., the root node in the execution tree. At this point, the entire infrastructure deployment is suspended and every VM is waiting for permission to continue. The Macrostep Debugger registers a root collective breakpoint for the infrastructure in the Neo4j database.

Depending on the mode of the debugging session, either the user or the Macrostep Debugger instructs one of the waiting VMs to continue its contextualisation. The selected VM continues its deployment, reaches/hits its next local breakpoint, the breakpoint script is executed again and the contextualisation of the VM is suspended again. The infrastructure at this point has reached another collective breakpoint.

At each collective breakpoint, the *Specification evaluator* component of the Controller evaluates the state of each virtual machine and the global state of the infrastructure. Further information regarding specification evaluation can be found in Section VII.

As the *contextualisation* of the virtual machines are developing, the execution tree is built by adding more and more nodes. Each node in the execution tree stores the following properties:

1) name of the infrastructure
2) collective breakpoint ID
3) previous collective breakpoint ID
4) process states
5) collected data
6) node type
7) instance IDs
8) exhausted flag
9) the result of specification evaluation

Collective breakpoints in the tree are associated with a unique ID (2) upon creation. Additionally, each node refers to its predecessor breakpoint (3). The Macrostep Debugger relies on these IDs to keep track of the infrastructure instance's current position in the execution tree.

The process states (4) property defines the current local breakpoint number for each virtual machine where they are suspended when reaching the node. It is stored by using the process names as keys and the values are the list of breakpoint numbers for each instance of the process named in the key. For example, the property with this value {"client": [1,2], "server": [2]} means that there are two instances of the client, where the first instance has been suspended at its first local breakpoint, while the second instance is suspended at its second local breakpoint. The server has one instance blocked at the second local breakpoint. The next property in the list, called *collected data* (5) stores every information collected and received from the virtual machines at the time of reaching the breakpoint.

The Macrostep Debugger is responsible for determining the type of a node (6) in the execution tree: "*alternative*" if there are more than one unfinished contextualisation processes or "*deterministic*" if there is only one such process. In case a collective breakpoint is the last collective breakpoint of an execution path, meaning there are no viable contextualisation processes to continue, then the collective breakpoint will be flagged as "*final*".

Infrastructure instances that traversed a node in the execution tree will be listed in the node's instance IDs (7) property. The exhausted flag (8) is primarily used for alternative breakpoints, indicating if every execution path starting from that node was fully traversed. Accordingly, if the root's exhausted flag is set to true, then the whole execution-tree was fully traversed.

Each collective breakpoint contains the result of specification evaluation (9). This contains the result of process level evaluation and global level evaluation.

Whenever a collective breakpoint is hit, the Macrostep Debugger performs the creation and linking of appropriate nodes in the execution tree as needed. When a new collective breakpoint has been created in the Neo4j database, a continue command is issued to one of the waiting, unfinished virtual machines, either by the user or the Macrostep Debugger depending on the mode of the debugging session.

This whole process is continued until all virtual machines have finished their contextualisation. At each collective breakpoint, the Macrostep Debugger checks the execution tree of the infrastructure, updating, creating and attaching new collective breakpoints if necessary. The Macrostep Debugger eventually detects that infrastructure deployment has been finished, once every virtual machine has finished its contextualisation process. It then instructs Occopus or Terraform to destroy the infrastructure instance.

The above described procedure represents the traversal of one execution path in the execution tree. The execution tree is built incrementally, i.e., another execution path can be traversed in the next turn.

C. DEBUGGING MODES

The Macrostep Debugger provides the following four different debugging modes:

- manual debugging,
- replaying,
- automatic debugging,
- free-run.

During *manual debugging* the user is prompted for choosing a virtual machine of the infrastructure to continue its execution. After selection, the virtual machine continues its contextualisation process and eventually the infrastructure instance will reach a new collective breakpoint, after which the user is prompted again. This is repeated until all virtual machines have finished their contextualisation, after which the Macrostep Debugger instructs the cloud orchestrator to destroy the infrastructure instance.

The *replay mechanism* is implemented through meta-breakpoints, meaning that a collective breakpoint ID is given at the start of the session by the user. The goal of the Macrostep Debugger in this mode is to coordinate the deployment of the virtual machines, breakpoint-by-breakpoint, to reach a collective breakpoint selected by the user. Initially, the debugger checks if the collective breakpoint exists in the infrastructure's execution tree. During replay, the execution path leading to the targeted collective breakpoint is followed. At each collective breakpoint hit, it is checked if the targeted collective breakpoint has been reached. If not, then the Macrostep Controller calculates which virtual machine's deployment needs to be continued in order to reach the next state leading to the targeted collective breakpoint.

During *automatic debugging*, the Macrostep Debugger itself decides which virtual machine will continue its deployment at each collective breakpoint. To select a virtual machine, the Controller component of the Macrostep Debugger creates an ordered list of the virtual machines using alphabetical ordering by name and ID. From this list, the debugger picks the first virtual machine that has not yet finished its deployment and instructs it to continue its deployment. This selection policy is applied at each collective breakpoint until the infrastructure instance has finished its deployment, after which the debugger instructs the cloud orchestrator to terminate the instance. At this point, the debugger backtracks the execution tree to the first non-exhausted, alternative collective breakpoint and initiates a replay session, targeting this collective breakpoint. The new infrastructure instance will eventually reach the targeted collective breakpoint and the Macrostep Debugger continues on a new, unexplored execution path, selecting the appropriate virtual machine. An alternative collective breakpoint will be flagged as exhausted if every execution path starting from that collective breakpoint is already explored. Automatic debugging essentially means the depth-first traversal of the execution tree, which continues until the root collective breakpoint itself is flagged as exhausted.

During *free-run mode*, virtual machines are not blocked upon reaching a local breakpoint, but rather are instructed to continue their contextualisation processes. In this mode, the debugger only notes/stores a timestamp of the virtual machines reaching their breakpoints. From these timestamps, an order of breakpoint hits can be constructed, which shows the execution path that was traversed. Free-run mode is useful for observing the behaviour of infrastructure deployment when no hindering factors are present. Conducting more and more free-runs eventually shows the tendencies in unhindered infrastructure deployment (e.g.: the traversal of one execution path is more likely than others).

V. ABSTRACTION OF ORCHESTRATION AND APPLICATION SCENARIOS

Our Macrostep Debugger first introduced in paper [9] was utilising the Occopus [10] tool in order to realise the orchestrator related low-level functionalities such as creation, contextualisation and shutdown of virtual machines in the target cloud. The low-level orchestration in our Macrostep architecture is designed to be outsourced to an existing orchestrator tool in order to rely on more complex functionalities and to speed up the development of the overall architecture. This design goal is straightforward in such environments, and we gained much advantage of it.

As every *orchestrator tool*, Occopus has also its limitations in terms of functionalities, supported clouds or maturity. At the beginning we have chosen Occopus as we had the full control over this solution and know-how. In order to overcome these limitations as well as to get rid of the dependence on one single orchestrator tool, we decided to generalise the interface of our Macrostep Debugger towards the orchestrator tool. Introducing this capability has opened up the way towards the capability to utilise any current or future cloud orchestrator tool. Once the interface has been redesigned to introduce an abstraction and the notion of orchestration handler has been added to the architecture, the Macrostep Debugger became independent of the underlying orchestrator tool.

The *orchestration handler* is intended to hide the details on controlling (instantiating, shutting down and invoking various methods of) the underlying orchestrator and to show a generalised interface towards the upper layers of the Macrostep Debugger in its architecture. With this additional feature, it is possible to add further cloud orchestrators serving the requirements of our Macrostep Debugger. To showcase the pluggability of the underlying orchestrator, we implemented the integration of the Terraform orchestrator tool with the Macrostep debugger.

Terraform is a cloud orchestration tool, developed by HashiCorp, which provides similar features like Occopus (e.g. instantiation, contextualisation, scaling and shutdown functionalities) over (a set of) the virtual machines in a cloud. Compared to Occopus, Terraform cannot be handled as a service, it does not expose a REST API, instead it is a command-line tool. This requires the orchestrator handler to cover this gap of functionality.

Since Terraform CLI can only be used through CLI commands, the orchestration handler in Macrostep debugger launches Terraform as a child process with standard inputs and outputs redirected and instructs Terraform via commands to complete necessary operations. To create an infrastructure instance, the Macrostep Debugger starts Terraform in the root module with the command *terraform apply -auto-approve*. As a result of this command, Terraform starts deploying the infrastructure. Tearing down the deployed infrastructure can be performed by issuing the command *terraform destroy -auto-approve*.

The Macrostep Debugger architecture is composed of several different components, such as a Neo4j graph-database, orchestrators such as Occopus and Terraform, infrastructure descriptors, cloud platforms etc. Most of these components can be accessed through endpoints and APIs, use various configurations and descriptors.

In order to make the *configuration and setup* of the debugging simplified, the Macrostep Debugger relies on so-called scenario descriptors. Scenario descriptors help in integrating all the application and debugger related settings into one entity. There are many settings such as the cloud infrastructure that will be tested (essentially the *debugee*), orchestrator related information, APIs and endpoints, Neo4j related information. Furthermore, the specification of states and conditions - which the infrastructure has to reach and fulfill - are also contained here. These settings are finally stored in a YAML file.

The detailed list of settings/configuration stored in a scenario descriptor is as follows:

- the name of the scenario,
- the type of the cloud-orchestrator,
- endpoint of the cloud-orchestrator,

- location of descriptor(s),
- Neo4j endpoint,
- Neo4j authorization information,
- a specification of states that the infrastructure has to fulfil,
- and a global specification with predicates.

During any kind of debugging session (replay, manual etc.), the Macrostep Debugger uses information stored in the scenario descriptor to initiate the deployment or tear-down of an infrastructure instance, to build the execution tree and to check if the virtual machines of the infrastructure instance has reached a specified state. The structure of the scenario descriptor is shown in Code 1.

The type of the orchestrator can either be "Occopus" or "Terraform". When the orchestrator is "Occopus", it is handled as a RESTful service, so an http URL points to the endpoint of orchestrator specified under "url". In the case of Terraform, the value of "local" is specified for the key "url" since Terraform is not handled as a service. The location of orchestrator specific descriptor files under the key "infra_file" defines the location of the infrastructure descriptor file in case of Occopus, or a working directory storing the descriptor file in case of Terraform. Orchestrators will utilize these to build infrastructure instances upon a request from the Macrostep Debugger.

Code 1. Structure of scenario descriptor.

```
scenario_name: [user defined name of the scenario]
orchestrator:
  type: [type of orchestrator i.e. occopus/terraform]
  url: [location of the orchestrator endpoint]
  infra_file: [location of infra descriptor file(s)]
execution-tree:
  host: [endpoint of Neo4j]
  user: [username for accessing Neo4j]
  password: [password for accessing Neo4j]
specification:
  [name of the variable]:
    vm: [virtual machine name]
    name: [name of the variable on the given VM]
    group: [all/any/none]
    expected: [expected value and operation]
  statements:
    - [logical expression]
```

Under the *execution_tree* section, all the information related to storing the execution tree inside the Neo4j tool is defined. Neo4j endpoint ("host") specifies a bolt protocol URL where the Neo4j graph database is available. To access the Neo4j database, username and password are required. Based on these three parameters, the debugger can access the Neo4j database and can create collective breakpoints, macrosteps, or can simply run the necessary queries needed for replaying or for automatic/manual debugging.

The section *specification* of the scenario descriptor structure contains specifications related to variables and their values found in a virtual machine. The debugger queries the value of the variable on a given virtual machine and performs checking described under "expected" subsection with values and operators. For further details about how these specifications are evaluated, please see Section VII.

The developments above altogether aimed to decrease the dependency on a single orchestrator and to introduce the generalisation of the orchestrator interface of the Macrostep Debugger. The developments towards other directions are going to be detailed in the next sections.

VI. AUTOMATIC BREAKPOINT GENERATION

Previously, Cloud-init contextualization files had to be manually instrumented, meaning that the insertion of local breakpoints was the responsibility of the user. However, this can be a tedious and error-prone operation, especially in case of longer contextualization files. To automate this process, a breakpoint generating component (or simply *Breakpoint generator*) was developed, which is responsible for generating instrumented Cloud-init files.

According to the concept of macrostep-based debugging described in Section III, macrosteps have to be pure, meaning that any communication-related command is also the last command of a macrostep. In effect, this means that local breakpoints have to be inserted right after these communication-related commands. In the case of Cloud-init contextualization files, the breakpoint generator component of the debugger has to find all communication-related commands, and then it has to insert a local breakpoint.

The current breakpoint generation technique relies on finding commands in the *runcmd* section of the Cloud-init files which contain a private IP-address. The assumption is that these commands indicate communication points between two virtual machines managed by the cloud-orchestrators. A local breakpoint has to be inserted after these communication points.

In order to find such commands, the *Breakpoint generator* relies on the so-called macro feature of Occopus, and the variable interpolation feature of Terraform. In both cloud orchestrators, these features serve the purpose of creating dynamic and more flexible contextualisation files. In Occopus, one such macro is called *getprivip()*, which can be used to inject the private IP-address of a given virtual machine into the Cloud-init file. In Terraform, users can insert user-defined variables, and then these variables will be interpolated in the final Cloud-init file. Both in the case of Occopus and Terraform-based infrastructures, it is assumed that the presence of certain macros and variable interpolations indicate communication with other virtual machines, making them communication-related commands.

In case of Occopus-based infrastructures, the Breakpoint generator is parsing for commands containing the *getprivip()* macro in the *runcmd* part of the Cloud-init files, while in the case of Terraform-based infrastructures, the Breakpoint generator is looking for commands which interpolate any variable containing *private_ip* in their name (e.g.: private_ip_server, private_ip_client).

In the first step during breakpoint generation, the Breakpoint generator collects the commands which contain the mentioned macros or variable interpolations. Then, the Breakpoint generator inserts a local breakpoint (script call) after each of these commands. In addition, the Breakpoint generator inserts a local breakpoint at the start and at the end of the Cloud-init file. The last local breakpoint has the additional argument "last". Thanks to this, the debugger will know that this local breakpoint is the last one in the contextualization process of the virtual machine, treating the virtual machine contextualisation as a finished process thereafter. At the very least, two local breakpoints will be present in an instrumented Cloud-init file, one at the very start and one at the very end of the contextualization process. Additional local breakpoints will be inserted after the above-mentioned communication points. An example Cloud-init before breakpoint generation can be seen in Code 2.

Code 2. Cloud-init original version.

```
runcmd:
- sudo apt-get -y install mariadb-client
- sudo mysql -u "root" "-pubuntu"
    -h "{{getprivip('srv')}}"
    -e "USE mstepdb;
    INSERT INTO demo (vm_name)
    VALUES ('{{variables.message}}');"
```

After breakpoint generation, the resulting instrumented Cloud-init is shown in Code 3.

Code 3. Cloud-init instrumented version.

```
runcmd:
- /tmp/MSTEP_BP.sh first
- sudo apt-get -y install mariadb-client
- sudo mysql -u "root" "-pubuntu"
    -h "{{getprivip('srv')}}"
    -e "USE mstepdb; INSERT INTO demo (vm_name)
    VALUES ('{{variables.message}}');"
- /tmp/MSTEP_BP.sh last last_bp
```

The above example contains one *getprivip()* macro (a communication point with another virtual machine), and therefore a local breakpoint is inserted after it. However, it coincides with the last breakpoint. In effect, two local breakpoints will be present in the instrumented Cloud-init file.

Although the current breakpoint generation technique provides a basis for more advanced procedures, it has its limitations. A limitation in the case of Terraform, is that only the interpolation of variables that contain *private_ip* in their name are detected. If a private IP-address is inserted into a command trough a variable that does not contain *private_ip* in their name (e.g.: a variable named *srv_ip_addr*), the breakpoint generator component can not detect it, and will not insert a local breakpoint. We expect from the user to follow the suggested naming convention or manual breakpoint insertion is still possible in this situation.

VII. SPECIFICATION EVALUATION

Evaluating the internal state of an infrastructure can be a cumbersome process [26], since this usually means verifying if each virtual machine was deployed according to the expectations of the user. This includes checking if the necessary processes, services, files, etc. are running, existing or have a specific content. In this section, we introduce the specification evaluation feature of the Macrostep Debugger, that helps users to automate this process and to detect the situation when an infrastructure deployment went according to their expectations.

As mentioned in Section IV, the local breakpoint script creates a JSON structure which contains information about the internal state of the virtual machine. It includes information related to identification, as well as user-defined information in the form of key-value pairs, whose value usually have the output of a shell command (e.g.: the number of records in an SQL database, the number of lines in a file, the content of a file, etc.). This JSON structure is then forwarded to the Macrostep Debugger, which stores this information in the Neo4j database.

In a scenario descriptor, users can create a specification which the infrastructure has to meet in order for the deployment to be considered satisfying. Users can do so by defining "global" variables, as well as statements that are made up of these "global" variables. Statements are logical expressions and can contain operators "*and*" or "*or*". An example specification can be seen in Code 4.

Code 4. Description of specification.

```
specification:
  variables:
    numLinesInMyFile:
      vm: worker
      name: numberOfLines
      group: all
      expected: 60
    lastBp:
      vm: worker
      name: isLastBP
      group: any
      expected: true
    fileContentOK:
      vm: master
      name: fContent
      group: any
      expected: "contains OK"
  statements:
    - numLinesInMyFile and lastBp or fileContent
```

In the example shown in Code 4, the specification contains three variables. These are "*numLinesInMyFile*", "*lastBp*" and "*fileContentOK*". Each variable has several attributes:

- vm,
- name,
- group,
- expected.

The attribute called "*vm*" refers to a virtual machine name as seen by the Macrostep Debugger. The attribute called "*name*" refers to a user-defined key in the JSON structure that was collected (which has a value that can be tested against an expected value), essentially the name of a "local" variable on that virtual machine. The attribute called "*expected*" is the expected value or condition this "local" variable has to satisfy. It can be an integer, float, or boolean, in which cases equality is tested. If the attribute "*expected*" is a string, then more complex expressions can be specified (e.g. contains a string, less than or equal, between

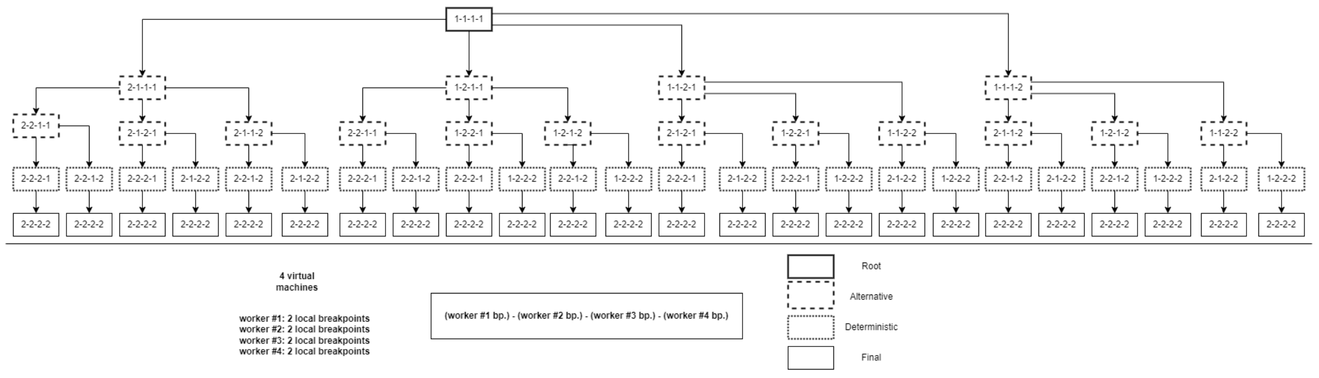

1-1-1-1
2-1-1-1
1-2-1-1
1-1-2-1
1-1-1-2
2-2-1-1
2-1-2-1
2-1-1-2
1-2-2-1
1-2-1-2
1-1-2-2
2-2-2-1
2-2-1-2
2-1-2-2
1-2-2-2
2-2-2-2
4 virtual machines
worker #1: 2 local breakpoints
worker #2: 2 local breakpoints
worker #3: 2 local breakpoints
worker #4: 2 local breakpoints
(worker #1 bp.) - (worker #2 bp.) - (worker #3 bp.) - (worker #4 bp.)
Root
Alternative
Deterministic
Final

FIGURE 7. Execution tree of an infrastructure with four virtual machines.

two number). Lastly, the attribute called "*group*" can either be "*all*", "*any*" or "*none*". Depending on this, either all, any, or none of the "local" variables with the given "*name*" on the virtual machines with the given "*vm*" have to satisfy the expectations.

After reaching a collective breakpoint, specification evaluation takes place. First, the Macrostep Debugger iterates over all "global" variables. At each variable, the debugger iterate over all virtual machines with the given "*vm*" name. At each corresponding virtual machine, the debugger evaluates if the given "local" variable with the specified "*name*" satisfies the expectations (i.e. its value is according to expectations). The result of this "local" variable evaluation will be a boolean value, true or false. After these "local" variable evaluations, the debugger continues with a list of true and false values. In the next step, the debugger decides if all, any, or none of these values in the list are true. If the attribute called "*group*" was "all", then all the values have to be true in order for the "global" variable to be true. If it was "any", any of the values have to be true in order for the "global" variable to be true. If it was "none", then none of the values in the list have to be true in order for the "global" variable to be true. Accordingly, the "global" variable will be either true or false.

After the Macrostep Debugger evaluated all "global" variables, it iterates over the statements. At each statement, it substitutes the "global" variable name with the evaluated value of the "global" variable. As a last step, the debugger evaluates the statement.

VIII. PARALLEL TRAVERSAL OF THE EXECUTION-TREE

Manually debugging the deployment of an infrastructure can be a tedious and exhaustive process. In order to speed up debugging, and ensure that each execution path is traversed, users can utilize the automatic debugging mode of the Macrostep Debugger. However, even with automatic debugging, the complete traversal of the execution tree of an infrastructure can take a considerable amount of time. In [9] we showed that the complete traversal of a relatively simple infrastructure containing four virtual machines can take several hours. It was a consideration at the time to help speed up the debugging process.

In order to achieve this, the Macrostep Debugger provides the option to traverse an execution tree in a parallel manner. This means that during automatic debugging, multiple and independent execution paths are traversed at the same time. At the start of an automatic debugging session, users can define a parallelisation level. This parallelisation level defines the maximum number of infrastructure instances that can be deployed at the same time.

At *startup of parallel automatic debugging*, the Macrostep Debugger waits until the first infrastructure instance reaches the root collective breakpoint. Then it checks if the root collective breakpoint is an alternative collective breakpoint, which means that there are multiple execution paths leading out from it. If that is the case, the Macrostep Debugger calculates the number of infrastructure instances to be started. The debugger tries to maximise parallelisation depending on the given parallelisation level and the number of alternative execution paths. Each of the running infrastructure instances will traverse different execution paths, creating necessary collective breakpoints on the way.

To investigate the advantages and behaviour of the parallel debugging feature of our Macrostep debugger, we have

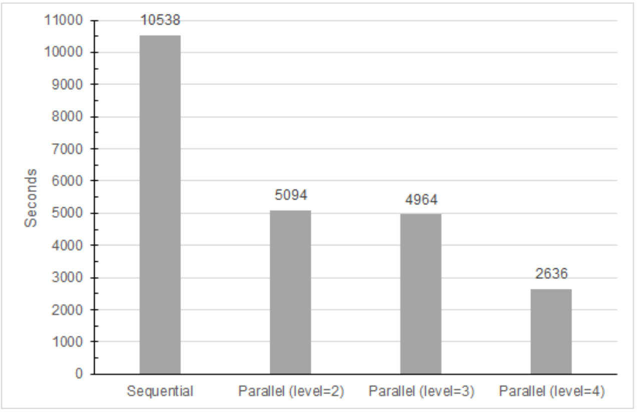


FIGURE 8. Required time for complete execution tree traversal during sequential and parallel automatic debugging.

selected two examples, the first one contains four virtual machines, while the second one is built from only three virtual machines but resulting a different category of execution tree.

Figure 7 shows the execution tree of the first example which deploys an infrastructure containing four virtual machines, eight local breakpoints and a total of twenty-four execution paths. The execution tree consists of four big subtree which owns the same amount of leaves i.e. execution paths. We consider such trees a well-balanced graph affecting the optimal number of parallelisation during deployment. Since our initial parallelisation solution only considers the second level of the tree during slicing, we assumed that for this example two or four instances are the most optimal in terms of efficiency.

Figure 8 shows the *speed-up* measured for the first example. Parallelisation levels were set to two, three and then four. At a parallelisation level of two, the time required to complete the parallel traversal of the execution tree was almost half compared to doing it sequentially. On a parallelisation level of four, the time required for a complete traversal was almost a quarter of a sequential traversal. At a parallelisation level of three, no significant speed-up was achieved compared to a parallelisation level of two. This is due to parallelisation being achieved at a "depth" of one from the root node. In the case of parallelisation level set to three, the first three subtrees of the execution tree are traversed at the same time (a total of eighteen execution paths, six for each subtree), and the fourth subtree is only traversed after that.

As a drawback, obviously resource consumption in cloud was twice, three times, and four times as much as compared to sequential debugging.

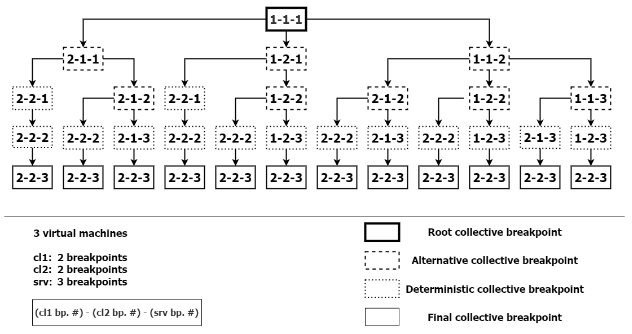


FIGURE 9. Execution tree of an infrastructure with three virtual machines.

Figure 9 shows the execution tree of the second example (unbalanced), an infrastructure containing three virtual machines, a total of ten local breakpoints, and twelve execution paths. In this case we investigated the situation when parallelisation level was set to three, meaning that at most three execution paths were traversed at the same time. In this case, the required time to complete a sequential traversal was *2869* seconds, parallel traversal was almost twice as fast compared to sequential at *1464* seconds, despite the parallelisation level being set to three. This is due to the execution tree being "unbalanced". In this case, two subtrees of the execution tree contained three execution paths each, while the third subtree contained six execution paths.

In summary, while sequential automatic debugging relies on depth-first traversal, parallel automatic debugging relies on the combination of depth-first and breadth-first traversal. The current parallelisation technique can provide significant speed-up in the cases of balanced execution trees. However, more refined parallelisation techniques are needed to be introduced in order to handle resource limitations (e.g.: quotas) and unbalanced execution trees (e.g.: parallelisation on deeper levels in the execution tree).

IX. DEMONSTRATION OF A REAL-LIFE USE-CASE

A. INTRODUCTION

In this section, we introduce a real-life use case in order to demonstrate and validate the usage of macrostep debugging for finding bugs in a reference architecture. Our selected use case is the cloud reference architecture version of the SLURM Workload Manager [41] which was developed by our laboratory for our ELKH (HUN-REN) users. During the development of the reference architecture, we successfully used the Macrostep Debugger to identify and reproduce erroneous timing conditions.

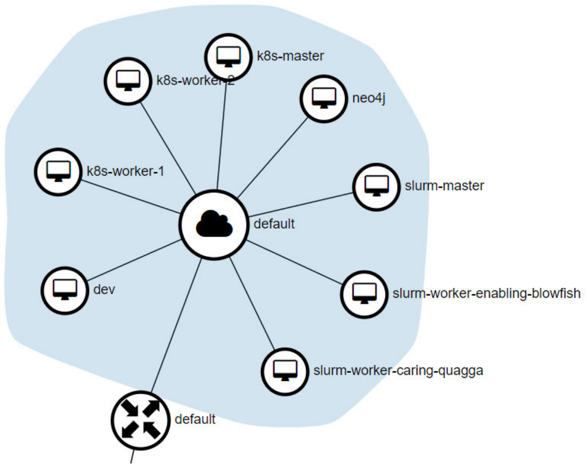


FIGURE 10. SLURM Master and SLURM Workers on the OpenStack topology page.

SLURM is a popular, well-known resource and cluster management system, that is capable of scheduling jobs on smaller and larger Linux-based clusters or on Supercomputers. It is open source, fault-tolerant and highly scalable. The cloud version of the SLURM reference architecture we developed and used for demonstrating our Macrostep debugger contains two types of virtual machines, a single Master and multiple Workers. Figure 10 shows the master and worker virtual machines (in a multi-tenant environment, please note the Kubernetes nodes as well) using the OpenStack network topology visualisation functionalities.

To understand its internal behaviour of the deployment, we give an insight about the installation and configuration steps of the Master and Worker nodes. The deployment of the reference architecture starts with the master setting up an NFS [42] Server, which the workers can use later on for sharing working directories storing files and configuration. The master then installs and configures Munge [43], a scalable authentication service for creating and validating keys and credentials. In the next step, it configures MariaDB [44], an open source fork of the MySQL [45] relational database. The deployment of the Master finishes with installing SLURM daemons and creating configuration files.

The deployment of Worker virtual machines follows similar steps during contextualisation. For the workers, the existence of Master is defined as a dependency, so they wait until a connection can be established with the master on port 6819. Workers then continue with configuring the NFS client in order to access the shared storage on the Master. In the next step, they install and configure Munge to synchronise credentials with Master. Finally, the workers install the SLURM client and create/modify configuration files. Since the reference architecture has been developed for Terraform, we were able to use it natively (thanks to the latest improvement described in Section V) as the cloud-orchestrator to build and manage the infrastructure.

Code 5. **Registration of workers.**

```
cp /storage/slurm/slurm.conf /etc/slurm-llnl/slurm.conf
echo "$(slurmd -C 2>&1 | head -n 1)
     NodeAddr=$(hostname -I | col1)" >>
     /etc/slurm-llnl/slurm.conf
su - ubuntu -c
     "cp /etc/slurm-llnl/slurm.conf
     /storage/slurm/slurm.conf"
```

On several occasions during the testing of the ready to use SLURM reference architecture, we have experienced faulty infrastructure deployment. The symptom of the incorrect deployment was the missing entries from the *slurm.conf* configuration file on Master, which caused the Workers to be lost from the cluster. Ideally, this file should contain records from all workers, however in rare situations (especially when the amount of worker nodes was higher, such as above ten) the configuration file missed the entries from a worker. Upon closer inspection of configuration file manipulation, we identified three lines (see Code 5) in the worker Cloud-init file which are responsible for updating the configuration i.e. registering the workers.

The *slurm.conf* configuration file is stored on the master virtual machine in a directory shared with all the workers by NFS. It is used (among others) to store relevant information about the worker virtual machines. Upon deployment of the infrastructure, this file is created by the master and later the workers - during their contextualisation - insert their own record into the file to inform the master and all other workers about their existence.

B. PREPARATION

In order to find the reason of incorrect deployment, we utilised the Macrostep Debugger, creating a scenario descriptor, instrumenting the necessary Cloud-init files and running an automatic debugging session. We used a smaller (scaled-down) infrastructure with one master and two workers since we believed that errors originated from incorrect timings can be easily tracked through a scaled-down version of the reference architecture due to the nature of macrostep debugging [31]. As a result, we developed a scenario descriptor (shown in Code 6) to set up the debugger.

Code 6. **SLURM scenario descriptor.**

```
scenario_name: scen_terra_slurm_elkh
execution-tree:
  host: [endpoint of Neo4j]
  password: [username for accessing Neo4j]
  user: [password for accessing Neo4j]
orchestrator:
  type: terraform
  infra_file: terraform_slurm\terraform_openstack
  url: local
specification:
  variables:
    numLinesEqual2:
      vm: slurm-worker
      name: numberOfLines
      group: all
      expected: 2
    lastBp:
      vm: slurm-worker
      name: isLastBP
      group: all
      expected: true
  statements:
    - numLinesEqual2 and lastBp
```

The content of sections *scenario_name*, *execution-tree* and *orchestrator* are straightforward and has been introduced in details in Section V. However, it is worth to investigate the content of the section called *specification*. It contains two variables, namely *numLinesEqual2* and *lastBp*. Variable *numLinesEqual2* is evaluated to True when both worker virtual machines detect exactly two records in the *slurm registry* file. At the same time, the variable *lastBp* is only evaluated to True if both workers have reached their last local breakpoint. Finally, the *statements* section (in Code 6) contains an expression that needs to be evaluated. The expression *numLinesEqual2 and lastBp* meant that we consider a deployment correct when both variables, i.e. *numLinesEqual2* and *lastBp* are True.

Code 7. **Instrumented cloud-init file.**

```
runcmd:
- /tmp/MSTEP_BP.sh mstep_start
- ...
- cp /storage/slurm/slurm.conf /etc/slurm-llnl/slurm.conf
- /tmp/MSTEP_BP.sh slurm_conf_copied
- echo "$(slurmd -C 2>&1 | head -n 1)
    NodeAddr=$(hostname -I | col1)" >>
    /etc/slurm-llnl/slurm.conf
- su - ubuntu -c
    "cp /etc/slurm-llnl/slurm.conf
    /storage/slurm/slurm.conf"
- ...
- /tmp/MSTEP_BP.sh last last_bp slurm_installed
```

To simplify/scale-down the execution tree generated by the Macrostep Debugger we did not include the master in the debugging process, we only instrumented the Cloud-init file of the Worker.

The instrumented Cloud-init file on the Workers contained three local breakpoints, one at the beginning of the contextualisation process, one after the first step of the worker registration and one local breakpoint at the very end of the contextualisation. The shortened version of the instrumented worker Cloud-init file is shown in Code 7.

Here, the preparation of the SLURM reference architecture has finished. In the next phase, the debugging process will happen to find the reason for the incorrect deployment.

C. DEBUGGING

After instrumenting the necessary *Cloud-init* file (Code 7) and importing the *scenario descriptor* (Code 6), we utilised the Macrostep Debugger and started an automatic debugging session. During this session, the execution tree was automatically generated. For each execution path, the deployment of the reference architecture has started, the breakpoints were hit, and the debugger has driven the execution of the deployment to follow an execution path that is different from the previously scanned ones. As a result, the execution tree has been built, shown in Figure 11.

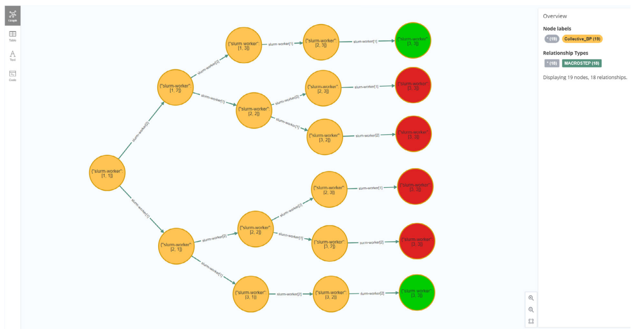


FIGURE 11. The execution-tree during the automatic debugging session, after complete traversal.

Altogether, there were six different execution paths that the debugger has traversed. The complete execution-tree with red leaf nodes indicates collective breakpoints where the given statement in the specification was evaluated to *False*. Leaf nodes, where the given statement in the specification was evaluated to *True*, are shown in green.

In cases of execution paths leading to the top-most and bottom-most leaf nodes (seen in green in Figure 11), we experienced that the *slurm registry* was correct. We have also seen that each of the ‘‘local’’ variable *numberOfLines* on the worker virtual machines (showing the number of worker records read in the *slurm registry*) equalled to two.

However, execution paths leading to the middle four (red) leaf nodes, the *slurm registry* was incorrect and each of the ‘‘local’’ variable *numberOfLines* on the worker virtual machines equaled to one. To inspect the faulty execution lines, it is possible to execute the deployment of the reference architecture to a certain point in the execution tree with the help of the Macrostep Debugger. Utilising this feature, it became clear that in the case of faulty deployments the *slurm registry* was copied by both worker nodes to their local disk before writing the updated content back to the shared directory. As a consequence, this concurrent behaviour resulted in the loss of worker update, performed by the virtual machine that reached the end of its critical section earlier.

In summary, this use case showed that the Macrostep Debugger was able to reproduce erroneous timing conditions, helping us in confirming that missing records are the result of concurrent file access. The debugger also successfully showed execution paths which lead to deployments satisfying a given specification.

D. COMPARISON WITH FREE-RUNS

In section IV we described a debugging mode of the Macrostep Debugger, called *free-run* mode. During free-runs, no active control happens, only timestamps are stored for monitoring purposes. These timestamps represent the points in time when the virtual machines hit their local breakpoints. By observing the order in which breakpoints were hit, it is possible to reconstruct the execution path the infrastructure instance traversed during the run. Although free-runs can be used to replicate faulty deployments, this debugging mode cannot provide systematic testing, execution is uncontrolled. The order in which the local breakpoints will be hit is practically randomised. Due to random selection of execution path, we observed that free-run debugging mode can only cover a small portion of the existing execution paths.

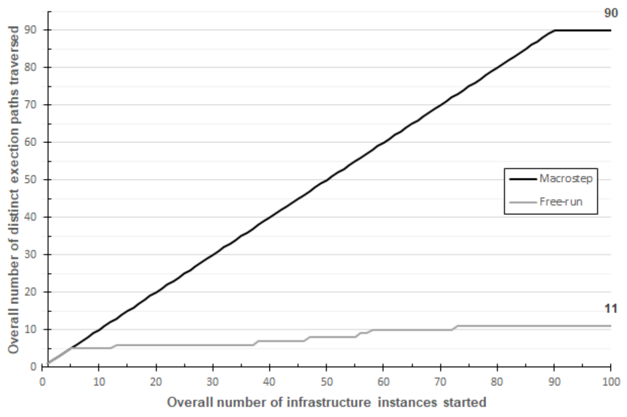


FIGURE 12. Distinct execution paths covered during free-runs and macrostep (three worker variant).

In order to demonstrate the advantage of automatic debugging mode (see subsection IV-C), we have prepared and fully tested two variants of the SLURM infrastructure. The first variant contained three workers, while the second consisted of four workers. The infrastructure and scenario descriptors were identical except for the expected number of workers in the *slurm registry*.

During macrostep-based automatic debugging/testing, we successfully discovered that the three worker variant has 90 different execution paths, out of which 6 satisfies the specification. In case of the four worker variant, there are

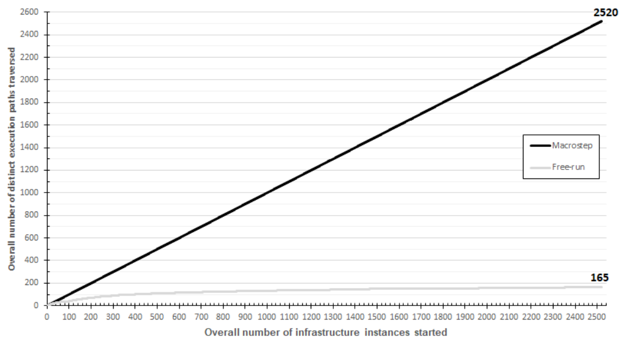


FIGURE 13. Distinct execution paths covered during free-runs and macrostep (four worker variant).

TABLE 3. Summary of evaluation metrics measured and calculated for the four worker SLURM deployments.

	Free-runs	Macrostep
Distinct execution paths existing	2520	2520
Distinct execution paths covered	165	2520
Overall test coverage (%)	**6.5**	**100**
Distinct satisfactory paths existing	24	24
Distinct satisfactory paths covered	24	24
Distinct satisfactory paths coverage (%)	**100**	**100**
Distinct unsatisfactory paths existing	2496	2496
Distinct unsatisfactory paths covered	141	2496
Distinct unsatisfactory paths coverage (%)	**5.6**	**100**
Total number of instances started	2520	2520
Total number of satisfactory deployments	2041	24
Total number of unsatisfactory deployments	479	2496
Unsatisfactory deployment rate (%)	**19**	**99**

2520 different execution paths (*test cases*), out of which only 24 satisfies the specification.

In order to make a comparison with the free-run debugging mode, we had to perform at the very least 90 free-runs for the three worker variant and 2520 for the four worker variant. We chose to make 90 free-runs for the three worker, and 2520 free-runs for the four worker infrastructure in our experiment. We have chosen these numbers respectively, so that having the same resource utilization (infrastructure cost) comparison could be made between the two debugging modes.

TABLE 2. Summary of evaluation metrics measured and calculated for the three worker SLURM deployments.

	Free-runs	Macrostep
Distinct execution paths existing	90	90
Distinct execution paths covered	11	90
Overall test coverage (%)	**12.2**	**100**
Distinct satisfactory paths existing	6	6
Distinct satisfactory paths covered	6	6
Distinct satisfactory paths coverage (%)	**100**	**100**
Distinct unsatisfactory paths existing	84	84
Distinct unsatisfactory paths covered	5	84
Distinct unsatisfactory paths coverage (%)	**5.9**	**100**
Total number of instances started	90	90
Total number of satisfactory deployments	82	6
Total number of unsatisfactory deployments	8	84
Unsatisfactory deployment rate (%)	**8.8**	**93.3**

Figures 12 and 13 show overall execution path coverage for the three and four worker infrastructure variants respectively. In case of the three worker infrastructure (see Figure 12), the collected monitoring information showed that during free-runs only 11 out of the 90 different execution paths were covered. In the case of the four worker infrastructure (see Figure 13), a total of 165 distinct execution paths were traversed during free-runs. In comparison, macrostep-based automatic testing was able to cover all execution paths in both cases.

We started 10 additional instances in free-run mode, and 10 additional instances in macrostep mode for the three worker variant, which are visible in Figure 12 (instances no. 91 to 100). It can be seen that the additional instances that were run in the macrostep mode did not cover any new execution paths, meaning that the *Macrostep* curve plateaued at 90 distinct execution paths. Additionally, it can be seen that the instances that were run in free-run mode, did not cover any additional distinct execution paths. This meant that the *Free-run* curve plateaued at 11 distinct execution path.

Table 2 shows summarized data from free-runs and automatic testing in case of the SLURM infrastructure containing three workers. As mentioned before, out of the 90 distinct execution paths, 6 were satisfactory while the remaining 84 execution paths were unsatisfactory. Both free-runs and macrostep-based debugging were able to cover all satisfactory paths, a rate of 100% for both test modes. However, free-run mode was only able to cover 5 distinct unsatisfactory paths (a rate of 5.9%), while automatic debugging was able to test all 84 of them (a rate of 100%). Overall, macrostep-based debugging was able to test all distinct execution paths (an overall test coverage of 100%), while in the case of free-runs, overall test coverage reached only 12.2% after 90 instances started.

In case of free-run mode, the total number of satisfactory deployments vastly outnumbers that of unsatisfactory deployments (which have a rate of 8.8%), which is as expected, since free-runs simulate deployments when no blocking mechanisms are enabled. In the case of automatic debugging it is the opposite, a high rate of unsatisfactory deployments are observed. This is due to the low number of distinct satisfactory execution paths (a total of 6), and since each infrastructure instance traverses a separate test case (execution path), automatic debugging will inevitably yield higher unsatisfactory deployment rate (93.3%).

Table 3 shows summarized data for the four worker infrastructure. For this variant, a total 2520 different execution paths existed, out of which only 24 were satisfactory. Automatic debugging was able to cover all 2496 unsatisfactory and all 24 satisfactory test cases, a rate of 100% for both. In comparison, while free-run mode was able to cover all satisfactory execution paths (a rate of 100%), it was able to cover only 141 unsatisfactory test cases (a coverage of 5.6%). Overall, automatic debugging was able to cover all execution paths, while free-runs only managed to cover 6.5% of them. In the case of the four worker variant (similarly to the three worker variant), free-runs have a lower unsatisfactory deployment rate (19%), while macrostep-based testing shows a 99% rate in this aspect. This is due to the reasons mentioned before.

Using previous information, the data in Table 2 and Table 3, an observation can be made. With the increasing number of workers (or virtual machines in general), the overall number of *test-cases* (or distinct execution paths) increases, too. In case of a two worker infrastructure there were 6 (see Section IX-C), in case of a three worker variant 90, and in case of a four worker variant, there were a total of 2520 different execution paths. While the number of VMs increases linearly, the number of execution paths increases exponentially.

To summarize, utilizing macrostep debugging method and active control shows clear advantages compared to traditional free-runs. Using the automatic debugging mode, each individual infrastructure instance is able to traverse a separate and distinct execution path, which eventually culminates in linearly increasing test coverage until it reaches 100%, in case total testing is required.

X. CONCLUSION AND FUTURE WORK

The paper introduces our latest results on applying the macrostep debugging technique for cloud orchestration and might be considered as the second major step towards a novel, generic, automated debugging framework for clouds.

In our first paper [9] we successfully identified the theoretical background including processes, breakpoints, collective breakpoints, and execution trees in the context of cloud orchestration. In order to validate our work, a prototype was developed relying on the Occopus orchestrator and the Neo4j graph database tools. We introduced the basic components and operation of the prototype as well as the manual-/auto-debugging, replaying, visualisation and query functionalities of the tool. The validation of the experimental framework has been started with use case and measurements with numerical results. Our mechanism was able to handle the non-deterministic behaviour of the cloud environment in terms of the unpredictable relative speed of resources (including virtual machines, networks, and I/O devices). Besides the reproducible errors, the described mechanism allowed us to explore the state space systematically and help in the detection of more possible erroneous situations with higher coverage.

Regarding the current work presented in this paper, we automated the instrumentation, i.e., the placing of the local breakpoints into the cloud-init files of the orchestrated nodes in the cloud infrastructure. The debugger prototype has been generalised further by handling and supporting the most widespread cloud orchestrator, the Terraform tool. The described parallel version of execution tree traversing may significantly improve the usability of the debugger tool in complex use cases. Moreover, the automatic correctness evaluation of consistent global states [32] at each macrostep was also studied and implemented during our research. New use cases have been evaluated to start demonstrating and categorising the possible errors: our debugging experiences concerning the deployment of the widely used SLURM cluster were detailed in the paper. Therefore, in our cloud debugger the fundamental macrostep mechanisms are enhanced at each level (approach) of distributed debugging:

- for reproducible behaviour ('Trace, replay and debugging' level): automated instrumentation and placing breakpoints for macrosteps,
- for analysis of alternative paths ('Integrated testing, active control and debugging' level): accelerated, parallel traversing based on macrosteps,
- for evaluation of correctness properties ('Predicate detection, active control and debugging'): deployment-wide, global predicate evaluation at each macrostep.
- for generalization these new methods towards the most widespread cloud orchestration tools.

Besides the presented achievements, we started studying the maintenance phase of cloud infrastructures, including new modelling and steering mechanisms towards possible suspicious or erroneous situations in the generated execution tree during the debugging stage of the development cycle. The first experiences have been published recently in [30], where the introduced smart functionalities rely on graph-based machine learning (GNNs) or autoencoders, and the training data sets can be generated by formal modelling and simulations.

In our research agenda, the next step is to integrate the results of our past and ongoing work into one unified framework, and develop further some crucial and challenging functionalities, e.g. new debugger mechanisms for cloud continuum (including edge devices), and support for higher level and more advanced specification languages, e.g. allowing temporal logic expressions [32].

The impact of our results is expected to be wide, since cloud computing has become a cornerstone of a large variety of research and innovation activities: the European Open Science Cloud (EOSC) [46] initiative and the Hungarian ELKH (HUN-REN) Cloud [47] are two prominent examples. The Occopus orchestration tool is a part of commercially supported MiCADO [4] framework, and commercial clouds are supported by Terraform, i.e. the possible impact of our results is even wider (for example, Occopus has been applied in the manufacturing sector as well [48]). Another way of impact is related to the reference architecture (or blueprint) concept [49] where the reliability and portability might be improved of such complex architectures by using our approach, and to be disseminated and exploited in strategic research infrastructure programs of the EU, such as ESFRI SLICES [50].

REFERENCES

[1] S Bhardwaj, L Jain, and S Jain, "Cloud computing: A study of infrastructure as a service (IAAS)," *Int. J. Eng. Inf. Technol.*, vol. 2, no. 1, pp. 60–63, 2010.

[2] M. Caballer, I. Blanquer, G. Moltó, and C. de Alfonso, "Dynamic management of virtual infrastructures," *J. Grid Comput.*, vol. 13, no. 1, pp. 53–70, Mar. 2015.

[3] R. Dukaric and M. B. Juric, "Towards a unified taxonomy and architecture of cloud frameworks," *Future Gener. Comput. Syst.*, vol. 29, no. 5, pp. 1196–1210, Jul. 2013.

[4] A. Ullah, H. Dagdeviren, R. C. Ariyattu, J. DesLauriers, T. Kiss, and J. Bowden, "MiCADO-edge: Towards an application-level orchestrator for the cloud-to-edge computing continuum," *J. Grid Comput.*, vol. 19, no. 4, p. 47, Dec. 2021.

[5] J. Cunha, J. Lourenco, and V. Duarte, *Debugging of Parallel and Distributed Programs*. Hauppauge, NY, USA: Nova Science Publishers, 2001, pp. 101–136.

[6] K. Fatema, V. C. Emeakaroha, P. D. Healy, J. P. Morrison, and T. Lynn, "A survey of cloud monitoring tools: Taxonomy, capabilities and objectives," *J. Parallel Distrib. Comput.*, vol. 74, no. 10, pp. 2918–2933, Oct. 2014.

[7] P. Kacsuk, R. Lovas, and J. Kovács, "Systematic debugging of parallel programs in DIWIDE based on collective breakpoints and macrosteps," in *Proc. Eur. Conf. Parallel Process.*, vol. 1685. Berlin, Germany: Springer, 1999, pp. 90–97.

[8] P. Kacsuk, G. Dózsa, J. Kovács, R. Lovas, N. Podhorszki, Z. Balaton, and G. Gombás, "P-GRADE: A grid programming environment," *J. Grid Comput.*, vol. 1, no. 2, pp. 171–197, 2003.

[9] B. Ligetfalvi, M. Emődi, J. Kovács, and R. Lovas, "Fundamentals of a novel debugging mechanism for orchestrated cloud infrastructures with macrosteps and active control," *Electronics*, vol. 10, no. 24, p. 3108, Dec. 2021.

[10] J. Kovács and P. Kacsuk, "Occopus: A multi-cloud orchestrator to deploy and manage complex scientific infrastructures," *J. Grid Comput.*, vol. 16, no. 1, pp. 19–37, Mar. 2018.

[11] *Occopus*. Accessed: Oct. 23, 2021. [Online]. Available: https://occopus.readthedocs.io/en/latest/

[12] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: Simple Linux utility for resource management," in *Job Scheduling Strategies for Parallel Processing*, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds., Berlin, Germany: Springer, 2003, pp. 44–60.

[13] J. Zhang, Z. Luan, W. Li, H. Yang, J. Ni, Y. Huang, and D. Qian, "CDebugger: A scalable parallel debugger with dynamic communication topology configuration," in *Proc. Int. Conf. Cloud Service Comput.*, Dec. 2011, pp. 228–234.

[14] J. Cai, J. Fei, X. P. Liu, H. Wang, Y. R. Wu, and S. Q. Zhong, "Remote debugging in a cloud computing environment," U.S. Patent 9 244 817, Jan. 26, 2016.

[15] *Microsoft Azure*. Accessed: Oct. 30, 2021. [Online]. Available: https://azure.microsoft.com/

[16] M. O. Ozcan, F. Odaci, and I. Ari, "Remote debugging for containerized applications in edge computing environments," in *Proc. IEEE Int. Conf. Edge Comput. (EDGE)*, Jul. 2019, pp. 30–32.

[17] M. Smara, M. Aliouat, A.-S.-K. Pathan, and Z. Aliouat, "Acceptance test for fault detection in component-based cloud computing and systems," *Future Gener. Comput. Syst.*, vol. 70, pp. 74–93, May 2017.

[18] P. Zhang, S. Shu, and M. Zhou, "An online fault detection model and strategies based on SVM-grid in clouds," *IEEE/CAA J. Autom. Sinica*, vol. 5, no. 2, pp. 445–456, Mar. 2018.

[19] K. Goossens, B. Vermeulen, R. V. Steeden, and M. Bennebroek, "Transaction-based communication-centric debug," in *Proc. 1st Int. Symp. Netw. Chip (NOCS)*, vol. 50, May 2007, pp. 95–106.

[20] *Snapshot Debugger*. Accessed: Aug. 30, 2023. [Online]. Available: https://github.com/GoogleCloudPlatform/snapshot-debugger

[21] X. Merino and C. E. Otero, "Microservice debugging with checkpoint-restart," in *Proc. IEEE Cloud Summit*, vol. 40, Jul. 2023, pp. 58–63.

[22] M. S. S. Quroush and T. Ovatman, "A record/replay debugger for service development on the cloud," in *Communications in Computer and Information Science*, V. M. Muñoz, D. Ferguson, M. Helfert, and C. Pahl, Eds., Cham, Switzerland: Springer, 2019, pp. 64–76.

[23] P. Sharma, S. Chatterjee, and D. Sharma, "CloudView: Enabling tenants to monitor and control their cloud instantiations," in *Proc. IFIP/IEEE Int. Symp. Integr. Netw. Manage. (IM)*, May 2013, pp. 443–449.

[24] Y. Gan, M. Pancholi, D. Cheng, S. Hu, Y. He, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of cloud debugging," in *Proc. 10th USENIX Conf. Hot Topics Cloud Comput.*, 2018, p. 13.

[25] H. Baek, A. Srivastava, and J. Van der Merwe, "CloudSight: A tenant-oriented transparency framework for cross-layer cloud troubleshooting," in *Proc. 17th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (CCGRID)*, May 2017, pp. 268–273.

[26] D. Cotroneo, L. De Simone, P. Liguori, and R. Natella, "Run-time failure detection via non-intrusive event analysis in a large-scale cloud computing platform," *J. Syst. Softw.*, vol. 198, Apr. 2023, Art. no. 111611.

[27] J. Manner, S. Kolb, and G. Wirtz, "Troubleshooting serverless functions: A combined monitoring and debugging approach," *SICS Softw.-Intensive Cyber-Phys. Syst.*, vol. 34, nos. 2–3, pp. 99–104, Jun. 2019.

[28] M. A. Gulzar, S. Wang, and M. Kim, "BigSift: Automated debugging of big data analytics in data-intensive scalable computing," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, New York, NY, USA, Oct. 2018, pp. 863–866.

[29] M. A. Gulzar, M. Interlandi, S. Yoo, S. D. Tetali, T. Condie, T. Millstein, and M. Kim, "BigDebug: Debugging primitives for interactive big data processing in spark," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng. (ICSE)*, New York, NY, USA, May 2016, pp. 784–795.

[30] R. Lovas, E. Rigó, D. Unyi, and B. Gyires-Tóth, "Experiences with deep learning enhanced steering mechanisms for debugging of fundamental cloud services," *IEEE Access*, vol. 11, pp. 26403–26418, 2023.

[31] R. Lovas and B. Vécsei, *Integration of Formal Verification and Debugging Methods in P-GRADE Environment*. Boston, MA, USA: Springer, 2005, pp. 83–92.

[32] J. Kovacs, G. Kusper, R. Lovas, and W. Schreiner, "Integrating temporal assertions into a parallel debugger," in *Proc. Eur. Conf. Parallel Process.*, B. Monien and R. Feldmann, Eds., Berlin, Germany: Springer, 2002, pp. 113–120.

[33] *Terraform*. Accessed: Aug. 22, 2023. [Online]. Available: https://www.terraform.io

[34] *Cloud-Init: The Standard for Customising Cloud Instances*. Accessed: Oct. 23, 2021. [Online]. Available: https://cloud-init.io/

[35] J. Webber, "A programmatic introduction to Neo4j," in *Proc. 3rd Annu. Conf. Syst., Program., Applications, Softw. Humanity*, Oct. 2012, pp. 217–218.

[36] *Amazon Web Services*. Accessed: Oct. 30, 2021. [Online]. Available: https://aws.amazon.com/

[37] E. Luchian, C. Filip, A. B. Rus, I.-A. Ivanciu, and V. Dobrota, "Automation of the infrastructure and services for an openstack deployment using chef tool," in *Proc. 15th RoEduNet Conf., Netw. Educ Res.*, Sep. 2016, pp. 1–5.

[38] V. Sobeslav and A. Komarek, "Opensource automation in cloud computing," in *Proc. 4th Int. Conf. Comput. Eng. Netw.*, W. E. Wong, Ed., Cham, Switzerland: Springer, 2015, pp. 805–812.

[39] M. Owens, *The Definitive Guide to SQLite*. New York, NY, USA: Apress, 2006.

[40] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč, "Foundations of JSON schema," in *Proc. 25th Int. Conf. World Wide Web*, Apr. 2016, pp. 263–273.

[41] *Slurm Reference Architecture*. Accessed: Sep. 2, 2023. [Online]. Available: https://git.sztaki.hu/science-cloud/reference-architectures/slurm/-/tree/v0.1.0

[42] *Network File System Protocol Specification*, document RFC1094: NFS, Sun Microsystems, 1989.

[43] *Munge: Munge Uid 'n' Gid Emporium*. Accessed: Sep. 3, 2023. [Online]. Available: https://dun.github.io/munge/

[44] *Mariadb Server: The Open Source Relational Database*. Accessed: Sep. 3, 2023. [Online]. Available: https://mariadb.org/

[45] *Mysql*. Accessed: Sep. 3, 2023. [Online]. Available: https://www.mysql.com/

[46] A. V. D. Almeida, M. M. Borges, and L. Roque, "The European open science cloud: A new challenge for Europe," in *Proc. ACM Int. Conf. Technol. Ecosyst. Enhancing Multiculturality*, 2017, pp. 1–4.

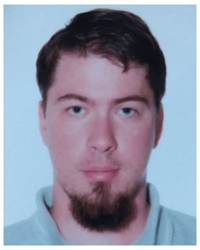
[47] *Elkh Cloud Portal*. Accessed: Nov. 9, 2021. [Online]. Available: https://science-cloud.hu/en

[48] S. J. E. Taylor, A. Anagnostou, N. T. Abubakar, T. Kiss, J. DesLauriers, G. Terstyanszky, P. Kacsuk, J. Kovacs, S. Kite, G. Pattison, and J. Petry, "Innovations in simulation: Experiences with cloud-based simulation experimentation," in *Proc. Winter Simulation Conf. (WSC)*, Dec. 2020, pp. 3164–3175.

[49] E. Nagy, R. Lovas, I. Pintye, Á. Hajnal, and P. Kacsuk, "Cloud-agnostic architectures for machine learning based on apache spark," *Adv. Eng. Softw.*, vol. 159, Sep. 2021, Art. no. 103029.

[50] *Esfri Slices Initiative*. Accessed: Aug. 30, 2023. [Online]. Available: https://www.slices-ri.eu/

JÓZSEF KOVÁCS received the B.Sc., M.Sc., and Ph.D. degrees in parallel computing, in 1997, 2001, and 2008, respectively. He is currently a Senior Research Fellow with the Laboratory of Parallel and Distributed Systems (LPDS), Institute for Computer Science and Control (SZTAKI), Hungarian Research Network (HUN-REN). His early research topics were parallel debugging and checkpointing, clusters, grids and desktop grid systems, and web portals. Recently, he has been focusing on cloud and container computing, especially on infrastructure orchestration and management. He gave numerous scientific presentations and lectures at conferences, universities, and research institutes in many places in Europe and outside. He is the author of more than 80 scientific publications including, conference papers, book chapters, and journals. He is a reviewer of several scientific journals and holds various positions at conferences.



BENCE LIGETFALVI received the B.Sc. and M.Sc. degrees from Óbuda University, in 2021 and 2023, respectively, where he is currently pursuing the Ph.D. degree with the Doctoral School of Applied Informatics and Applied Mathematics. He is a Research Assistant with the Laboratory of Parallel and Distributed Systems, Institute for Computer Science and Control (SZTAKI), Hungarian Research Network (HUN-REN). He is also an Assistant Lecturer with the John von Neumann Faculty of Informatics, Óbuda University. His primary research interests include cloud computing and debugging in cloud environments. His research work was awarded at the 36th National Scientific Students' Associations Conference and supported by the Hungarian Scientific Research Fund.

RÓBERT LOVAS received the Ph.D. degree in informatics from Budapest University of Technology and Economics (BME). He is currently the Deputy Director of the Institute for Computer Science and Control (SZTAKI), Hungarian Research Network (HUN-REN). He is a Habilitated Associate Professor and the Founder of the Institute for Cyber-Physical Systems, John von Neumann Faculty of Informatics, Óbuda University. His research and development experience in a wide range of application fields of distributed and parallel systems has been gained in various global, EU, and national collaborations with academic organizations, universities, and enterprises focusing on computational chemistry, numerical meteorological modeling, bioinformatics, agriculture, connected cars, and Industry 4.0. He has coordinated four EU projects (FP7, H2020, and Horizon Europe) and is responsible for managing national and pan-European research infrastructures as the Project Director of HUN-REN Cloud and an Executive Board Member of the EGI Foundation, which provides key assets to European Open Science Cloud. His latest cloud, big data, the IoT, and AI-related research achievements contribute to the recently launched Artificial Intelligence and Autonomous Systems National Laboratories. He is a member of the Committee on Information Science, Hungarian Academy of Sciences.