Digital Twins for Autonomic Cloud Application Management

Geir Horn, Rudolf Schlatte, and Einar Broch Johnsen

Abstract Cloud applications are distributed in nature, and it is challenging to orchestrate an application across different Cloud providers and for the different capabilities along the Cloud continuum, from the centralized data centers to the edge of the network. Furthermore, optimal dynamic reconfiguration of an application often takes more time than available at runtime. The approach presented in this paper uses a concurrent simulation model of the application that is continuously updated with real-time monitoring data, optimizing, and validating deployment reconfiguration decisions prior to enacting them for the running applications. This enables proactive decisions to be taken for a future time point, thereby allowing ample time for the reconfiguration actions, as well as realistic Bayesian estimation of the application's time variate operational parameters for the optimization process.

1 Introduction

A Cloud application is intrinsically a set of components distributed over different locations and infrastructures, hence it becomes challenging to maintain the application performance and stability over time as it may require constant attention of dedicated DevOps engineers. This challenge is not new and *Autonomic Computing* has been proposed as a solution [17]. The core concept is the Monitor, Analyse, Plan, Execute — with Knowledge (MAPE-K) [14] feedback loop continuously monitoring essential application metrics, and then plan and adapt the application to the application's current execution context. The concept has been used to build application management platforms for mobile computing [9], ubiquitous computing [11], and Cross-Cloud computing [12].

Department of Informatics, University of Oslo, P.O. Box 1080 Blindern, 0316 Oslo, Norway
e-mail: Geir.Horn@mn.uio.no, e-mail: rudi@ifi.uio.no,
e-mail: einarj@ifi.uio.no

A rational decision is in psychology and economy understood as the choice maximizing the *utility* of the decision maker [10]. Autonomic computing approaches therefore assume that the DevOp engineer's decisions can be replaced by a *utility function* that balances the different concerns and trade-offs implicit in the decision [18]. However, research has shown that it is hard for a DevOps engineer to formulate the utility function [8]. This barrier may be even higher as the number of configuration choices increases when the application can be deployed on heterogeneous computers that may be severely restricted in capacity, or when it is possible to deploy the application components in variants targeting different hardware accelerators like Graphics Processing Units (GPUs), Tensor Processing Units (TPUs), and Field-Programmable Gate Arrays (FPGAs).

An alternative to model the utility is to simulate the candidate deployments and pick the one that best satisfies the high-level goals and intentions of the DevOps engineer. This approach resembles Data Farming, *i.e.*, simulation experiments where the model parameters are varied across the simulations to compare the simulation output with measurements from the real system, thereby identifying the unknown underlying system parameters [13]. Data farming has been used similarly to make better decisions by complementing and improving the quality of the data mining of the observed 'big data' [24]. However, autonomic Cloud application management based on the MAPE-K loop continuously monitors the operational parameters of the application and its execution context. This allows the simulation model to be causally connected with the real word through the measurements. Hence, the model becomes a Digital Twin (DT) [4] of the running application. Instead of executing multiple simulations to identify unknown system parameters, as one would do for data farming, the novelty here is to execute multiple simulations for various reconfiguration options using the DT model as a part of making optimized decisions for the application configuration for the application's current execution context.

The optimized autonomic Cloud application management and the concept of a DT are discussed in Section 2 as a background to understand the problem and its complexity. A prerequisite for the proposed DT approach to work is that the executable modelling language used to implement the DT is accurately representing the salient application characteristics. The Abstract Behavioral Specification (ABS) [15] language fulfils the DT requirements, and Section 3 introduces the ABS language and a generic ABS model for Cloud applications. Section 4 briefly discusses the context of our proposed solution, and Section 5 concludes the paper.

2 The Cloud Application and its Digital Twin

2.1 Cloud Application Modelling

Cloud computing is first and foremost a *business model* where the infrastructure necessary to execute an application will be rented on demand. Permanently renting

resources will in the long run be more costly than owning the resources, and hence this implies that Cloud computing is for applications that are infrequently needed, or to cover intermittent peak loads of permanently running applications. It is not necessary to manage automatically short-lived applications. Hence, the type of applications considered here are the ones that are running for a significant time with variable resource needs over the execution time and where the resources needed permanently should be owned by the application owner with the variable needs for resources covered by rented resources offered by the Cloud providers.

A Cloud application is therefore a *distributed* application: It is minimally divided between the part running on the private infrastructures and a variable part running in the Cloud. The application can be seen as an orchestration of *software components*, where the term *component* is understood as an abstraction for an application building block [20]. The application components can be either software modules, applications, or packages; web services or Cloud platform functions; or data sources or sinks.

Any computing system is fundamentally about data transformation. This property transfers to the individual application components. Each component collects some input data, does some manipulation of the data, and produces some form of output data. These three phases can be seen as conducted in a strict sequence, and each phase has a starting time and a duration. This simple data flow delay model is illustrated in Figure 1. It is assumed that the autonomic application management system monitors four metrics for each component: The component's start time and the duration of the three phases. Note that this fundamental view also applies for stream data processing modules where the module's functionality should be further refined into component functions processing a part of the data input flow producing a part of the output flow. Finally, the three phases are also valid irrespective of the programming paradigm and philosophy used to create the component.

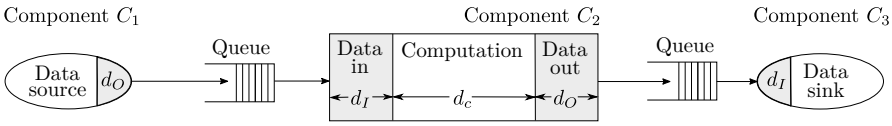

Fig. 1 The simple data flow model where there are intrinsic stochastic delays in each component caused by data input, d_I, data computation, d_c, and data output, d_O. Each component has individual probability distributions for these delays. Delays in the virtual work queues are included in the downstream component's input delay.

Hence, without lack of generality the application can be modelled as a set of components, each fulfilling the three phases of the data transformation. The time taken in each of the three phases will consequently depend on the data location and data size, and on the vertical scalability parameters like the number of cores given to a component or the amount of memory it has. In addition, it is necessary to model the data flow of the application since the input data for a component comes from one or more other computing components or data sources, and the output data flows

into other computing components or data sinks. This is illustrated in Figure 2. Horizontal scalability, *i.e.* how many instances there are of a particular component type, will affect the overall application completion time, the *makespan*, but the number of components will not affect how long it takes one component to finish its data transformation phases. There are already many options for describing this topology model of the application covering the type of components, the application's data flow, the components' computational requirements and constraints, their scalability constraints, and the application's DevOps engineers' deployment and operational goals [6].

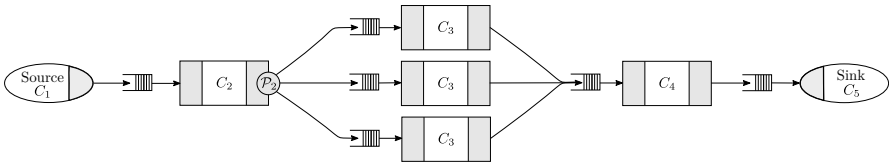

Fig. 2 Data flows where data is split on multiple instances of a component type, here C_3, require policies $\mathcal{P}$ on the upstream components deciding how the data is forwarded to the instances of the downstream component type(s). Popular policies are *round-robin*, *random*, or *broadcast* where the same data goes to all downstream instances.

Assuming that such a domain specific model of the managed application is available, it can be converted automatically into a DT simulation model representing the three fundamental phases of computation for each component, and the component's output-to-input connectivity. In general, all duration parameters are stochastic. In the best case, the DevOp engineers may have an idea for the *prior* duration time distributions to be included in the domain specific model. Otherwise, one may assume some standard distributions, typically Gaussian for their analytic tractability and well understood use in non-linear regression [23]. It is however difficult for the application's DevOp engineers to know the parameters for these duration distributions. For instance the duration of the computation phase will depend on data size and possibly also on the content of the data, and so it is hard to estimate the mean duration and the variance of the computation time *a priori*.

The DT component model will therefore maintain secondary probability distributions for the parameters of the duration distributions used in its three execution phases. These secondary distributions may better be defined by the application's DevOp engineers. The parameters of these secondary *prior distributions* will be recursively refined based on the monitored metric values from the running application using the Markov Chain Monte Carlo (MCMC) method, and in the case no prior secondary distribution is given it can be estimated from available monitoring information using an empirical Bayes approach [7].

The scalability parameters of a DT component will be set by the optimizer of the autonomic application management platform. Some of these *vertical* scalability parameters will directly affect the performance of the component, *e.g.*, the number of cores or the amount of memory or the use of a hardware accelerator. Changing the

component performance will directly affect the secondary distributions. Thus, the DT component model will maintain and update one set of secondary distributions per hardware configuration option available for the component.

The location of the components will affect the parameters for the two communication phases of the component model. Consider a computing component reading data from a data component. In this case the duration of the data input phase will be shorter if the data component is located in same region of the same Cloud provider as the computing component. The minimum duration will be when both components are located in the same data centre, but there is no way for the application management to control the Cloud provider's allocation of loads to its data centres within a given region. Hence, even for the same region of the same Cloud provider the mean duration of the data input phase will be a *random variate*, and this variation may be orthogonal to the variation caused by the size of the data set exchanged. Thus, the DT model should also maintain and update location specific secondary distributions for the parameters of the communication delay distributions.

2.2 Digital Twins

A DT is a digital replica of an underlying system, often called the Phyiscal Twin (PT) [26]. The DT is connected to its PT in real-time through continuous data streams such as sensor measurements at different locations and by other ways of collecting data. This turns the DT into a *live replica* of the PT, with the purpose of providing insights into its behaviour, and clearly distinguishes a DT from, *e.g.*, a standard simulation model.

A DT is commonly seen as an architecture with three layers: the *data layer* with, *e.g.*, Computer Aided Design (CAD) drawings and sensor data, an *information layer*, which turns these raw data into structured data, and an *insight layer*, which applies different analysis and visualization techniques to the structured data. The analysis techniques of the insight layer can be classified as follows: The DT is typically able to compute an approximation of how the PT acts in a given scenario (*simulation* or "what-happened" scenarios), or to estimate how the PT will behave in the future based on historical and current data (*prediction* or "what-may-happen" scenarios). By configuring the parameters of the different models, the DT may analyse the consequence of different options on future behaviour (*prescription* or "what-if" scenarios).

In the context of Cloud application management, we aim for a DT which can offer prescription. The data layer of the DT will initially consist of information about the configuration of the heterogeneous Cloud environment in which the application will execute, the resource profile of different locations such as bandwidth, memory and processing capacity, and the data flow topology of the application. The DT will continuously add information to the data layer about the software components to be orchestrated, including the duration of executing the different phases of the data transformation associated with a software component on a given location in the

various Cloud environments. Thus, the main technology needed to implement the data layer of a digital twin is a solution for timed data streams, such as a time-series data base.

The information layer will assimilate data from the data layer into a model which combines the application topology with the current configuration of the heterogeneous Cloud environment, transforming recorded or default durations to location-independent execution costs and gradually improving the precision of costs by learning from previous executions of the application workflow. Thus, the information layer combines information about the application topology and the resources provided at different locations in the current configuration of the Cloud environment. A challenge in the construction of DTs is to find a representation at the information layer which supports the required level of analysis. For a DT supporting Cloud application management, we propose to represent the information layer as an executable model in which different deployment decisions may be easily expressed and efficiently explored through simulation. The model should have a formal semantics which makes it transparent for the Cloud application management how to configure the parameters and understand the results from the executable model. This turns the DT into a tool for optimized Cloud application management.

2.3 Optimized Cloud Application Management

Each application component, $C \in \mathbb{C}$, has a set of requirement attributes, $\mathbb{A}_C$. These typically specifies the resource necessary for the component to perform as expected. Thus, a requirement attribute can be the number of cores useful for the component ranging from a minimum number to a maximum number, or it can be the amount of memory, or the Cloud providers that can be used, or the geographic location of the hosting data centre, or the number of copies or instances of the component the application can successfully exploit. The requirement attributes can be continuous or discrete. Each value of a requirement attribute value, $a_{C,i}$, is taken from the attribute's *domain*, $a_{C,i} \in \mathbb{A}_{C,i}$, for all the requirement attributes $i = 1, \ldots |\mathbb{A}_C|$.

A *configuration* of a component is an assignment of values to all its requirement attributes. The different ways a component C can be configured is its *variability space*, which is the Cartesian product of its attribute domains, $\mathbb{V}_C = \mathbb{A}_{C,1} \times \cdots \times \mathbb{A}_{C,|\mathbb{A}_C|}$. The configuration of the Cloud application is a flattened vector, $\boldsymbol{c}$, of the configurations of *all* the components of the application. The variability space of the application is therefore the Cartesian product of the variability spaces for its components, $\mathbb{V} = \mathbb{V}_1 \times \cdots \times \mathbb{V}_{|\mathbb{C}|}$.

The ultimate goal of the DevOps engineers or the autonomic platform managing the application will be to find *the best* configuration, $\boldsymbol{c}^*(t_k)$ for the application's current *execution context*, $\boldsymbol{\theta}(t_k)$, which is a vector of the metric values of the monitored application at the time point t_k. The traditional applications of autonomic computing assumes that the goals and preferences of the application's DevoOps engineers can be captured as a *utility function* from the variability space to the unit

interval, $U : \boldsymbol{c} \in \mathbb{V} \mapsto [0, 1]$. Finding the best configuration given the current execution context means, most likely, solving a non-linear mixed continuous-discrete optimization problem

$$\boldsymbol{c}^* (t_k) = \underset{\boldsymbol{c}(t_k) \in \mathbb{V}}{\operatorname{argmax}} U (\boldsymbol{c}(t_k) \,|\, \boldsymbol{\theta}(t_k), \boldsymbol{\phi}) \tag{1}$$

where $\boldsymbol{\phi}$ is a set of fixed parameters for the utility function family. This problem must be solved subject to a set of deployment constraints whenever there is a new execution context, *i.e.*, whenever any of the monitored values changes. The metric values can change abruptly and frequently, for instance as application users come and go, leaving little time for the optimization of the configuration before the next context change happens. The latter point is remedied by checking only the constraints, or the Service-Level Objectives (SLOs), of the optimization problem when a new measurement arrives, and then stay with the existing configuration if it remains feasible under the altered execution context.

The Bayesian approach of the DT defined in Section 2.1 allows the optimization to be done entirely based on the SLOs defined for the application. The same solver used for the mathematical programme (1) can be used. Searching for an optimal configuration, it will generate a sequence of candidate application configurations, $\boldsymbol{c}\,(t_k)$, feasible for the application's current execution context. One may then run a set of parallel simulations using the DT application model for each candidate configuration where the components randomly draw delay times from the delay distributions using the Maximum Likelihood Estimates (MLEs) of all delay distribution parameters. Alternatively, the secondary distributions, updated on each measurement from the running application, allow precise confidence intervals to be given for the parameters of the delay distributions. The simulation can be executed using these worst case limits for the distribution parameters. For both alternative ways of selecting the delay distribution parameters, one computes the relevant application performance metrics indicating the goodness of the overall application performance indicators under the chosen set of distribution parameters for a candidate configuration from the ensemble of its DT simulations.

Evaluating the SLOs for the computed application performance metric values will indicate if a configuration candidate is likely to remain feasible, or if a new and better configuration must be deployed for guaranteeing the continued feasibility of the application. Only the feasible configuration candidates will be retained and scored according to some high level goals set by the DevOps engineers like 'least deployment cost' or 'minimal reconfiguration from the currently running configuration', and the candidate with the highest score can be selected as the next deployment configuration.

The approach can be illustrated with a small data farming application with three components: A dispatcher component sending off data parallel jobs to a set of worker components, and one result processing component receiving the output of the workers, see Figure 2. All jobs must be completely processed by a given deadline. Assume that the execution time delay distribution of a job on a worker is Gaussian, $N(\mu, \sigma)$. At the 95% confidence level, the delay value will be in the interval

$\mu \pm 1.96\sigma$. From the confidence intervals of the secondary distributions at the same level of significance one has that $\mu \in [\mu^-, \mu^+]$ and $\sigma \in [\sigma^-, \sigma^+]$, and so the worst case bound on the execution time of one job at one worker is $\mu^+ + 1.96\sigma^+$. However, as multiple workers are processing jobs in parallel, there could be a queue of results for the downstream component causing additional global delays, and possibly this component could end up as a bottleneck in the application. For this particular example, one could calculate the queuing delays using queueing theory, but given that the arrival distribution of the results on the last component is unknown and dependent on the processing delay distributions of the worker, it is in general impossible to model this system analytically and assess if the deadline is met for a given application configuration. As the autonomic application management must work for any application, simulating the DT is the only way to collect realistic performance statistics for the managed application.

3 Abstract Behavioral Specification model as the Digital Twin

3.1 The Abstract Behavioral Specification Language

The ABS language combines implementation-level specifications with verifiability, high-level design with executability, and formal semantics with practical usability. ABS is a concurrent, object-oriented, modelling language with a functional layer with algebraic datatypes and side-effect-free functions. Of particular interest for this paper is *Real-Time ABS* [16], which additionally features a time and resource model: going beyond purely behavioural specifications, ABS models can specify time elapsed on a logical clock at certain points in the program logic, as well as resource usage when executing certain code paths. This resource usage will again influence the observed logical time for model execution. Instead of a full introduction to the language, which can be found, *e.g.*, in the language manual[1], this section maps ABS language features to the concepts of Cloud application modelling discussed in Section 2.1.

ABS as a language is based on the Actor model [3], with actors executing concurrently and communicating via asynchronous method calls. Method call results are returned via Future variables, which the caller can synchronize on. This naturally models the components of a Cloud application and their synchronous and asynchronous communication patterns.

Actors in ABS possess state in the form of fields bound to values. Computations on values (numbers, strings, boolean, actor references, futures, and user-defined algebraic data types) are expressed in a purely functional sub-language that is amenable to formal analysis. Depending on the desired level of detail of the Cloud application model, we can consider only the control flow of the data transformation steps mentioned in Section 2.1, or include manually abstracted or real computa-

[1] https://abs-models.org/manual/

tions in the model. In case the Cloud model has different, data-dependent control flow that needs to be modeled, typically some form of data will be included in the model. Multiple instances of components, and dynamic creation and tear-down of components, can be modelled by creating and releasing actors modelling these components. It is straightforward to model auto-scaling resource pools of such components.

As mentioned, the time consumed by a component's computation can be modelled via the logical clock implemented in Real-Time ABS. Elapsed time can be specified as constant or dependent on actor state, *e.g.*, as a function of the length of an input parameter to the computation. Logical time only elapses when explicitly specified; otherwise, computations are modelled to run "infinitely fast".

To model the deployment of the Cloud application components on machines with different resources, ABS implements a language feature called *Deployment Component* [16]. A deployment component serves as a location for one or more actors, and has a set of resources like computation speed, network bandwidth, memory, number of cores that it distributes among its actors. An actor can explicitly specify a resource need like computation or bandwidth for a step in its computation; that operation will take a certain amount of logical time depending on the amount of available resources.

Finally, ABS implements a *Model Application Programming Interface (API)* that allows to access the state of dedicated objects and call methods on these objects via Hypertext Transfer Protocol (HTTP) requests from outside the running model. Object state and method call results are returned in the JavaScript Object Notation (JSON) [21] data format. For simple visualizations and interactions with a running model, the model can serve HyperText Markup Language[2] (HTML) and JavaScript to a web browser and react to requests from that browser session.

3.2 Modelling Cloud Applications with ABS

The ABS features are closely aligned with the Cloud application modelling concepts, so converting those models into ABS will lead to understandable code, even when done automatically or semi-automatically. All of data-dependent delays, deployment-dependent delays and data transfer delays are directly represented at the ABS language level.

The component types shown in Figure 1 can be implemented as actors. Each component's queue will be modelled explicitly inside the ABS actor, to ensure the First In is the First Out (FIFO) semantics and to make the queue length and other attributes available in the model.

Simple component parameterization, *e.g.*, parameters for random distributions of data and computation sizes, can be implemented via local actor state. These param-

[2] https://html.spec.whatwg.org/

eters can be changed at run time by a dedicated monitor component, and accessed and stored from outside via the Model API during and after model execution.

More complex behavioral differences, *e.g.*, the data flow policies shown in Figure 2, can be implemented either via ABS traits as mix-ins of methods into class definitions or, in case policies should be adaptable at run time, via a method changing its behavior depending on local state. Again, policy state can be accessed from outside the model via the Model API.

The Cloud application topology can be implemented via ABS deployment components that model the spatial arrangement of components on machines with varying attributes. Some additional modeling or implementation work might be required if it becomes necessary to model complex machine topologies with non-trivial cost functions for transferring data between two given machines.

Data that needs to be persisted, *e.g.*, initial and computed random distribution parameters for each component, can be exported via the Model API. ABS includes read-only support for the SQLite database engine; this can be used for bulk initialization of components at model run-time. Behavior or parameter changes during a model run can be triggered via the Model API.

4 Discussion

There are many approaches available for a Cloud operator to schedule the incoming workload [1, 19]. It is important to note that the Cloud application management discussed in this paper takes an *application centric view*. In other words, there are no restrictions on new Cloud resources other than budget and time to acquire the needed virtual resources. The application components are allocated to the available virtual resources ignoring how the application's virtual resources are *scheduled* on the physical data centre hardware by the involved Cloud operator(s).

This paper has discussed the modelling of the DT using Bayesian estimation of the stochastic parameters of the involved distributions. The resulting DT model can be seen as resembling application modelling using a Bayesian Network (BN) [2]. BNs assume binary node states organized as a Directed Acyclic Graph (DAG), and have recently been used for applications with DAG dependencies to schedule the application components on a fixed number of heterogeneous resources [22]. The closest related approach to the ideas presented here is probably the modelling of planning problems as a dynamic BN in time and delay variables with direct sampling of the simulated BN to estimate the overall plan completion time (makespan) [5].

However, autonomic Cloud application management is not only about executing the application as quickly as possible, and the DT allows constrained optimization for multiple objectives on carefully selected adequate resources. The use of the ABS modelling language allows the extension of the DT refining the application model if needed to capture all details for validating various candidate application configurations and probabilistically simulating their performance indicators; some related case studies are listed in [25].

5 Conclusion

Constrained autonomic Cloud application management requires the optimization of the application configuration. This paper has presented the vision of using a Digital Twin (DT) based on the Abstract Behavioral Specification (ABS) language as a simulation model to assess the feasibility of various application configuration candidates given the application's Service-Level Objectives (SLOs). The proposed DT approach avoids the need for the cumbersome utility function definition normally required for autonomic computing, and allows better expressiveness for the application model thereby enhancing the optimization of the application's deployed configuration. This promising approach will hopefully soon be implemented in an autonomic Cloud application management platform to demonstrate its benefits for real world applications.

Acknowledgements

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871643 MORPHEMIC[3] *Modelling and Orchestrating heterogeneous Resources and Polymorphic applications for Holistic Execution and adaptation of Models In the Cloud*

References

1. A. R. Arunarani, D. Manjula, and Vijayan Sugumaran: Task Scheduling Techniques in Cloud Computing: A Literature Survey. Future Generation Computer Systems 91, 407–415 (2019). DOI: 10.1016/j.future.2018.09.014
2. Adnan Darwiche: Modeling and Reasoning with Bayesian Networks. Cambridge University Press (2009). DOI: 10.1017/CBO9780511811357
3. Agha, G.A.: ACTORS - a model of concurrent computation in distributed systems. MIT Press (1990)
4. Barbara Rita Barricelli, Elena Casiraghi, and Daniela Fogli: A Survey on Digital Twin: Definitions, Characteristics, Applications, and Design Implications. IEEE Access 7, 167653–167671 (2019). DOI: 10.1109/ACCESS.2019.2953499
5. Beaudry, E., Kabanza, F., and Michaud, F.: Planning for Concurrent Action Executions Under Action Duration Uncertainty Using Dynamically Generated Bayesian Networks. In: Proceedings of the International Conference on Automated Planning and Scheduling, pp. 10–17 (2010)
6. Bergmayr, A., *et al.*: A Systematic Review of Cloud Modeling Languages. ACM Computing Surveys (CSUR) 51(1), 22:1–22:38 (2018). DOI: 10.1145/3150227
7. Bradley P. Carlin, and Thomas A. Louis: Bayesian Methods for Data Analysis. Chapman and Hall/CRC (2008). DOI: 10.1145/3150227
8. Floch, J., *et al.*: Playing MUSIC — Building Context-Aware and Self-Adaptive Mobile Applications. Softw. Pract. Exper. 43(3), 359–388 (2013). DOI: 10.1002/spe.2116

[3] http://morphemic.cloud

9. Geihs, K., *et al.*: A Comprehensive Solution for Application-Level Adaptation. Softw. Pract. Exper. 39(4), 385–422 (2009). DOI: 10.1002/spe.900
10. Gilboa, I.: Rational Choice. MIT Press (2010). DOI: 10.1002/spe.900
11. Hallsteinsen, S., *et al.*: A Development Framework and Methodology for Self-Adapting Applications in Ubiquitous Computing Environments. Journal of Systems and Software 85(12), 2840–2859 (2012). DOI: 10.1016/j.jss.2012.07.052
12. Horn, G., and Skrzypek, P.: MELODIC: Utility Based Cross Cloud Deployment Optimisation. In: Proceedings of the 32nd International Conference on Advanced Information Networking and Applications Workshops (WAINA), pp. 360–367. IEEE Computer Society (2018). DOI: 10.1109/WAINA.2018.00112
13. Horne, G., and Schwierz, K.-P.: Summary of Data Farming. Journal of Systems and Software 5(1), 8–27 (2016). DOI: 10.3390/axioms5010008
14. IBM: An architectural blueprint for autonomic computing. White Paper Third Edition, p. 34. IBM (2005). DOI: 10.1016/j.jss.2012.07.052
15. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., and Steffen, M.: ABS: A Core Language for Abstract Behavioral Specification. In: Proceedings of the International Symposium on Formal Methods for Components and Objects (FMCO 2010). LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2010). DOI: 10.1016/j.jss.2012.07.052
16. Johnsen, E.B., Schlatte, R., and Tapia Tarifa, S.L.: Integrating deployment architectures and resource consumption in timed object-oriented models. Journal of Systems and Software 84(1), 67–91 (2015). DOI: 10.1016/j.jlamp.2014.07.001
17. Kephart, J.O., and Chess, D.M.: The Vision of Autonomic Computing. Journal of Systems and Software 36(1), 41–50 (2003). DOI: 10.1109/MC.2003.1160055
18. Kephart, J.O., and Das, R.: Achieving Self-Management via Utility Functions. Journal of Systems and Software 11(1), 40–48 (2007). DOI: 10.1109/MIC.2007.2
19. Kumar, M., Sharma, S.C., Goel, A., and Singh, S.P.: A Comprehensive Survey for Scheduling Techniques in Cloud Computing. Journal of Network and Computer Applications 143(12), 1–33 (2019). DOI: 10.1016/j.jnca.2019.06.006
20. Lau, K.-K., and Wang, Z.: Software Component Models. Journal of Systems and Software 33(10), 709–724 (2007). DOI: 10.1109/TSE.2007.70726
21. Pezoa, F., Reutter, J.L., Suarez, F., Ugarte, M., and Vrgoč, D.: Foundations of JSON Schema. In: Proceedings of the 25th International Conference on World Wide Web. WWW '16, pp. 263–273. International World Wide Web Conferences Steering Committee (2016). DOI: 10.1145/2872427.2883029
22. Pranab K. Muhuri, and Sajib K. Biswas: Bayesian Optimization Algorithm for Multi-Objective Scheduling of Time and Precedence Constrained Tasks in Heterogeneous Multiprocessor Systems. Applied Soft Computing 92(12), 106274 (2020). DOI: 10.1016/j.asoc.2020.106274
23. Rasmussen, C.E., and Williams, C.K.I.: Gaussian Processes for Machine Learning. MIT Press (2005). DOI: 10.7551/mitpress/3206.001.0001
24. Sanchez, S.M., and Sánchez, P.J.: Better Big Data via Data Farming Experiments. In: Advances in Modeling and Simulation: Seminal Research from 50 Years of Winter Simulation Conferences. Ed. by A. Tolk, J. Fowler, G. Shao, and E. Yücesan, pp. 159–179. Springer (2017). DOI: 10.1007/978-3-319-64182-9_9
25. Schlatte, R., Johnsen, E.B., Kamburjan, E., and Tapia Tarifa, S.L.: Modeling and Analyzing Resource-Sensitive Actors: A Tutorial Introduction. In: Damiani, F., and Dardha, O. (eds.) Coordination Models and Languages, pp. 3–19. Springer (2021). DOI: 10.1007/978-3-030-78142-2_1
26. Tao, F., Zhang, H., Liu, A., and Nee, A.Y.C.: Digital Twin in Industry: State-of-the-Art. Journal of Systems and Software 15(4), 2405–2415 (2019). DOI: 10.1109/TII.2018.2873186