Asynchronous and composable distributed co-simulation engine design for cloud-based industrial CAE software

Bohu Zhou, Jianwen Cao, Xuesong Wu, Yupeng Wang

Bohu Zhou, Jianwen Cao, Xuesong Wu, Yupeng Wang, "Asynchronous and composable distributed co-simulation engine design for cloud-based industrial CAE software," Proc. SPIE 12814, Third International Conference on Green Communication, Network, and Internet of Things (CNIoT 2023), 128142D (20 October 2023); doi: 10.1117/12.3010714

SPIE.

Event: Third International Conference on Green Communication, Network, and Internet of Things (CNIoT 2023), 2023, Chongqing, China

Asynchronous and Composable Distributed Co-simulation Engine Design for Cloud-based Industrial CAE Software

Bohu Zhou [ab], Jianwen Cao*[a], Xuesong Wu*[a], Yupeng Wang[a]
[a] Institute of Software, Chinese Academy of Sciences, Beijing, China;
[b] Email: zhoubohu20@mails.ucas.ac.cn
* Corresponding author: jianwen@iscas.ac.cn, xuesong@iscas.ac.cn

ABSTRACT

The industrial Internet can be seen as a combination of industrial software, sensors, networks and cloud platforms. Among them, industrial software is the key to data utilization while the cloud platform provides support for Industrial Internet. Through "visualization", "structuring", "modeling and simulation" and other means, industrial software can help solve the design and manufacturing problems of industrial products. But the difference between industrial Internet and traditional Internet lies in the different application scenarios and data characteristics, which also leads to the different emphasis of system architecture design. Considering the needs of high concurrency and huge data processing, the distributed system must provide high performance of communication and computing services. In addition, with the continuous expansion of the simulation model and the simulation scale, it becomes particularly significant to reduce the model development complexity and improve maintainability and flexibility. To make the distributed co-simulation engine more high-performance, we propose an asynchronous communication module as the underlying architecture of the distributed engine. Then combining the discrete event system specification and Actor model, a layered architecture has been proposed, which implements the modular programming of behavioral model by functional programming. Based on the architecture mentioned above, a distributed CAE cloud platform is proposed, which is hierarchical, modular, composable and reusable.

Keywords: Distributed co-simulation engine; Asynchronous communication; Composable; Actor-Oriented; DEVS; CAE; Cloud-based platform

1. INTRODUCTION

Industrial Internet and Industry 4.0 are prevailing topics in industry and academia. It makes an unprecedented impact on global industry development [1]. It is not only limited to the manufacturing industry, but also has a profound impact on other fields, such as medical health, agriculture, food industry, energy management, education, military and so on. Industry 4.0 not only brings the generation of new technologies, but also includes the recombination of many mature technologies. Modeling and Simulation is one of the key technology in industrial software, which can be used in the fields of enterprise planning and evaluation, industrial engineering, production chain management, industrial product experiment and verification [2].

Compared with traditional centralized and integrated industrial software, cloud-based industrial co-simulation software faces many problems [3] in the distributed scenario. The first is communication efficiency issues. In distributed network, the message transmission does not come "for free" and has certain costs. Hence communication should be treated as a computational resource and efforts must be made to use it wisely. Secondly, there is incomplete knowledge problem in distributed network, which means a processor has only a partial picture of the system and the ongoing activities. It is therefore difficult to coordinate a common activity. In addition, coping with failures and recoveries, timing and so on need to be solved. Therefore, it is significant to build a distributed industrial co-simulation architecture to handle these issues.

With the need for increasingly complex software systems and industrial model design, Model-Based Systems Engineering [4] (MBSE) was incorporated. Under the concept of modern software engineering and MBSE, the typical features of the complex system are typically Multi-Agent System, co-simulation, discrete event driven design. Hence it is vital to utilize the simulation method to the verification. Discrete Event Simulation Specification [5][6][7] (DEVS) proposed a complete modeling and simulation framework, which depicts the relationship between the entities in multi-scale. To make the same effect in digital world before construct in real world, we can automate the generation, composition and deployment based on the modular hierarchical concept in DEVS.

Third International Conference on Green Communication, Network, and Internet of Things (CNIoT 2023),
edited by Hongzhi Wang, Shiling Zhang, Proc. of SPIE Vol. 12814,
128142D · © 2023 SPIE · 0277-786X · doi: 10.1117/12.3010714

Based on the needs for industrial internet interaction, interconnection and complex system modeling, it is crucial to promote the industrial software to the lightweight cloud-based industrial platform. To maximize the utilization of the modeling and simulation tools, there comes up a paradigm called Modeling and Simulation as a Service (MSaaS) [8], which is expected to (1) provide scalable and composable facilities. (2) make underlying infrastructure and platform transparent to users. In this paper, we pay attention to asynchronization, resource usage, modularity, composability and so on. Then we implement a distributed co-simulation engine based on asynchronous communication, distributed composable technology, which applied to the cloud-based CAE platform to solve the issues of traditional single software mode [9].

2. DESIGN OF ASYNCHRONOUS MODULE

Futures and promises are a popular abstraction for asynchronous programming. In a broad sense, a future or promise can be thought of as a value that will eventually become available. In the rest of the section, we will give the specific design of the asynchronous module derived from promise/future concept, which comprise Event Loop, Reactor pattern, Dynamic Thread Pool.

2.1 Event Loop

In this paper, we divided the event loop into several stages, as shown in Figure 1 below, each of which has its own linked list of event queues to be processed. The first is the timer stage, which deals with various timing tasks; The second is the network I/O polling stage: this stage processes various network requests, inserts new requests into the event queue to be processed, checks the completed event queue, and executes the callback function of the completed network request; Finally, there is the asynchronous event processing stage: This stage maintains a linked list of asynchronous events such as thread pool tasks. In the polling process of the event macro loop, each stage has its own micro loop. In each micro loop, the queue is checked for pending events, and if so, the event is fetched and the appropriate action is performed; If not, proceed to the next stage; Finally, after the execution of all stages of the micro loop is completed, the next event is entered.

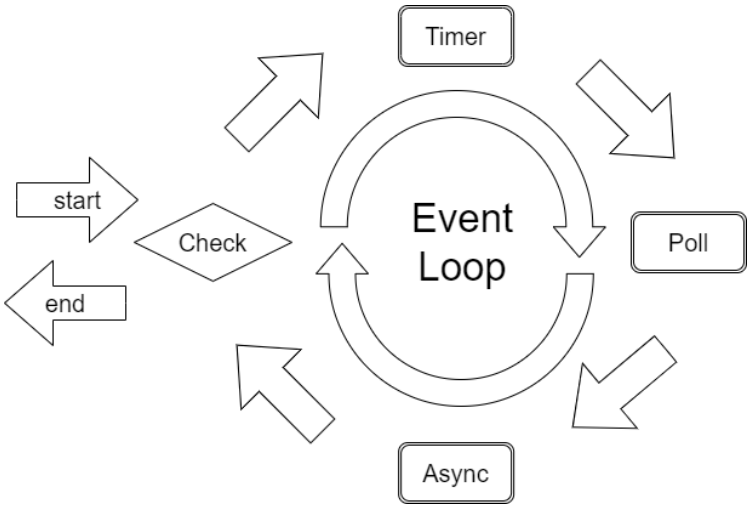

Figure 1. Design of the event loop.

2.2 Reactor Pattern

There are two important roles: Reactor and processor. Reactor listens to and acquires the request, creates a socket and registers the event through the I/O multiplexing [10] function. The underlying operation system function reads the data requested or writes the data. After the event completes, the corresponding callback event is invoked through the reactor loop, and the following business logic continues to be handled. If some service logic is time-consuming, the Reactor is assigned to a processor in order not to block the main thread. The processor transfers the task to the thread pool for the following operations. Because of the separation of Reactor from the processor, the processor does not need to care about the underlying I/O implementation details, allowing for higher concurrency and scalability. The reactor pattern is shown in Figure 2.

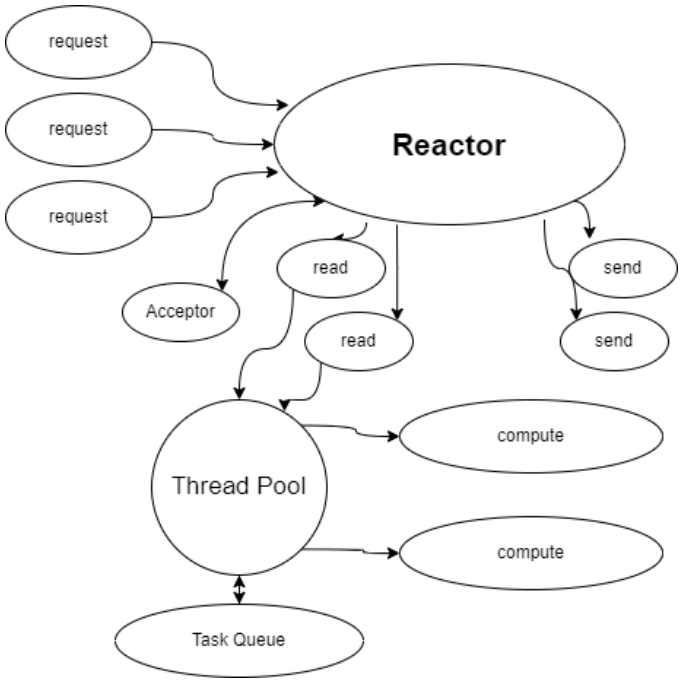

Figure 2. Design of reactor pattern.

Multi-Process Reactor

The multi-process Reactor server uses a Master/Slave design, in which each slave process maintains its own event loop. The Master process is responsible for accepting network requests and dispatching tasks in a round-robin manner. After the initialization the Master processor, it creates specific number of Worker processes. Then the worker process creates a new Reactor server, and monitor the incoming network request. After the connection is successful, the acquired socket will be transferred to the Worker process through inter-process communication by master process.

The Worker process handles specific request tasks. Each Worker will create a listening service to listen for information from the Master process. When receiving a socket from the Master process, the Reactor server created by the Worker process will handle the corresponding request by parsing the socket.

Connection Pool

Normally, a client needs a TCP connection to send a request, and then closes the connection after receiving a response. However, in order to achieve reliable data transmission, TCP adopts three-way handshakes before connection and four way handshake after termination, which is relatively time-consuming. The ability to reuse TCP connections and send multiple requests on the same connection provides a performance boost. Therefore, this section implements a module for pooling management of TCP connections. The pooling function mainly relies on the TCP Keep-Alive mechanism.

There are three main dictionary objects maintained in Connection Pool. *sockets* queue represents the socket currently in use, *free_sockets* represents the socket that is free, and *waiting_req* is the request that is waiting for the socket to be free. When the client sends a request, it checks whether there are free sockets in the connection pool. If there are free sockets, the socket is reused directly and inserted into the *socket*s queue. If there are no free sockets and the number of sockets does not reach the threshold, a new socket is created and used to process the request. If the above two conditions are not satisfied, the request is added to the *waiting_req*. When there is a free socket, the request is removed from the waiting queue for processing.

2.3 Dynamic Thread Pool

Multithreading technology can make full use of CPU resources, but the creation and destruction will bring great resource consumption, so in order to balance to improve the concurrency ability and improve resource utilization and reduce the response speed, usually use thread pool method to network programming. Thread pools are generally good for the following scenarios: (1) potentially blocking business logic (such as file reading and writing): The main thread of the event

loop can only handle relatively simple logic, if the thread gets stuck in time-consuming operations blocking subsequent requests, thus reducing the concurrency and throughput of the service. (2) CPU-intensive tasks: The execution of such tasks on the main thread will inevitably cause long-term thread blocking, resulting in the accumulation of new tasks and affecting the processing of new requests.

Thread pool model is widely used in distributed server applications. In order to maximize the performance, many problems concerning quality of service (QoS) need to be properly dealt with. For example, if the thread pool receives a large number of computation requests at the same time at any given moment, the quality of service may suffer significantly, such as an increase in response time. A common solution is to dynamically control the number of threads in the thread pool, adding threads to the pool during busy computing hours and reclaiming them during idle hours. But the number of thread pools is not the more the better: First, frequent thread creation and destruction can also reduce performance. Second, too many threads waste memory space and CPU, because the cost of switching threads back and forth during CPU scheduling is also an important factor. Third, too few threads do not make full use of CPU resources. Therefore, how to design the automatic expansion and shrinkage mechanism of thread pool management can significantly improve the system performance [11]. In this paper, a prediction model based on frequency is adopted to control the dynamic expansion strategy of thread pool, as shown in Figure 3 below.

In this paper, the newly opened thread regularly counted the frequency of receiving requests, predicted the future increase of requests to the thread pool by frequency changes and using EMA formula to predict the future frequency, and expected that there were enough threads in the thread pool for task processing. As requests decrease, the number in the thread pool slowly decreases depending on the expiration time.

$$EMA_t = \alpha \times P_t + (1 - \alpha) \times EMA_{t-1} \quad (1)$$

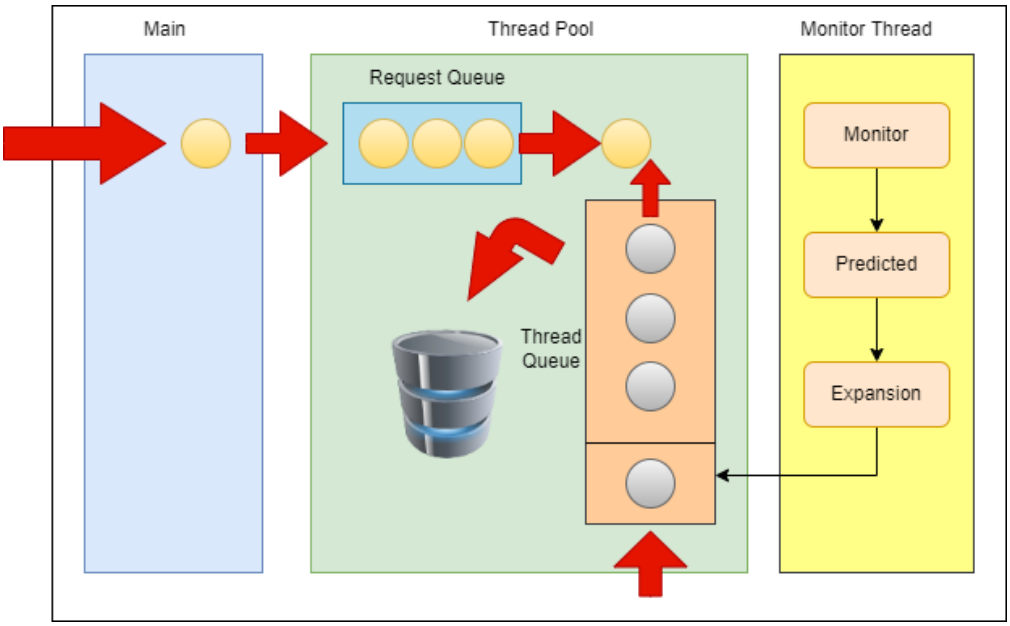

Figure 3. Dynamic thread pool architecture.

3. DESIGN OF COMPOSABLE DISTRIBUTED CO-SIMULATION MODULE

To meet the automated distributed composition, deployment and ever-changing modeling needs of users, the co-simulation module combined DEVS and Actor model [12] and took functional programming into the design pattern.

3.1 ActorDEVS Architecture

The entire ActorDEVS simulation service module is divided into two layers, as shown in Figure 4. The upper layer is the user control layer, which is used to provide various management services, and the bottom layer is monitored and controlled. The bottom layer is the simulation layer, which controls the interactions between actors through the DEVS specification. The lower layer is transparent to the modeling engineer, and only the upper layer provides user-relevant functionality.

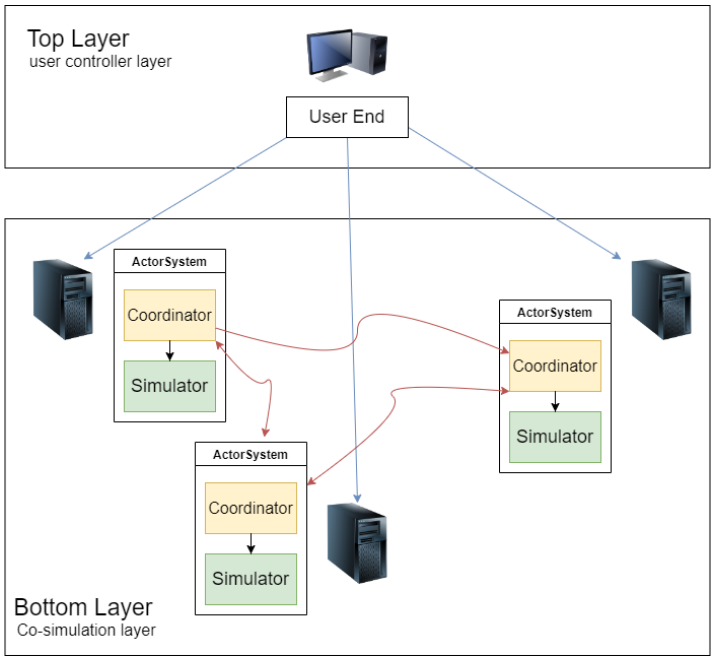

Figure 4. The separated frontend-backend ActorDEVS architecture.

In the bottom layer, there are three types of actors that matters: *CoordinatorActor*, *SimulatorActor* and *RouterActor*. The *CoordinatorActor* delegates the Coupled Model to control the life cycle of the coupled model and is responsible for monitoring the child nodes to ensure the availability of the service by using the coupling information in the coupled model. In *SimulatorActor*, Atomic models are delegated to perform specific behaviors. These Actors may be dispatched on the same machine, or they can be distributed on different physical machine manually or through load balancing strategies. When these actors try to send messages to each other, they send message through *RouterActor*. Each *RouterActor* maintains two dict objects called *local* and *remote*, which stores the key-value pairs of the components and their communication address. To improve the communication efficiency, we use UNIX Domain Socket rather than TCP when the components are dispatched on the same machine through local router.

The DEVS co-simulation is done through a central coordinator. The coordinator creates n simulation services over the internet. Each simulations service then generated m simulators that contains the DEVS model. Figure 5 shows the process. Once the simulation starts, the coordinator gets the time of the next event, and then executes the output function for each simulation service. Then the output function propagates. Propagation means that the coordinator takes the output of all the simulation services and sends it to other simulation services based on the coupling information. Finally, the state transition of each model is triggered; And it repeats.

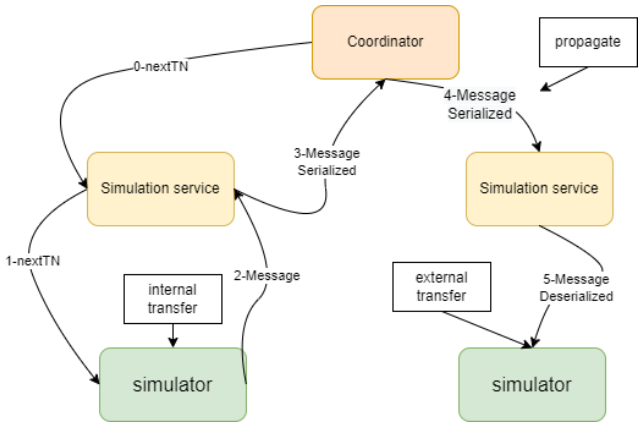

Figure 5. The co-simulation procedure.

3.2 Modular programming of behavior model

In this paper, we solved the problems of inheritance and polymorphism in OOP through the modularity and higher-order functions property in functional programming [13] (FP). We separated the DEVS entity structure and the DEVS behavior model, and then programming the composable behavior in a separate module, which combines the both characteristic of OOP and FP.

Among the traditional design patterns, there is a well-known State Pattern that seems to aid in the development of state management for finite state machines, making the code more maintainable. This pattern, however, focuses only on state. For external event transfer functions in DEVS, there is still an inconvenience when new events are introduced into the current entity and the behavior needs to be extended. Therefore, in combination with functional programming, we proposed a "event-state-pattern" which is shown in Figure 6. This pattern provides a combinable, pluggable and reusable state management module for the modular programming of DEVS behavior model.

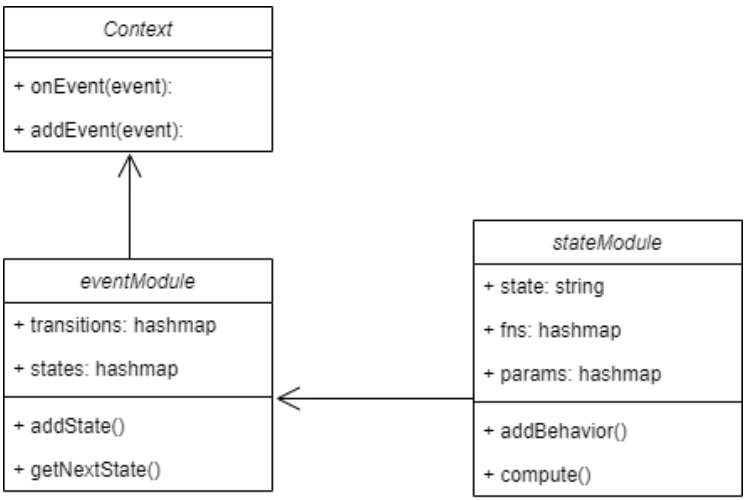

Figure 6. The event-state-pattern.

In the "event-state-pattern", we not only extracted the states out, but also extracted the events into a separate module. In detail, we instantiated a module object for each event or state in the finite state machine. Each object holds the key-value of the event-state mapping. Developer only needs to design the event and corresponding state and their callback functions when they are invoked in specific circumstance. Composing the functions through the principles of functional programming, developer only need to write a few functions or reuse existing functions to achieve modular, composable, extensible, pluggable, testable programming.

Under the rules of FP, there are two rules to follow:

(1) State is read-only: the only way to change state is to trigger the *compute* function of the state module, which executes the callback function passed in when developer design the state.

(2) State transfers are made by pure functions: To describe how the behavior model modifies state, the developer needs to write the *transition* function. So eventModule has a mapping *transitions* to index the state module (the mapping key-value pair is the stateModule instantiated state object and transition function). Then calculate the modified state variable with *getNextState*, and then design the target state after the current event is triggered in the current state by writing the *transition* function that takes the global state variable as an argument. The invoking procedure is shown in Figure 7.

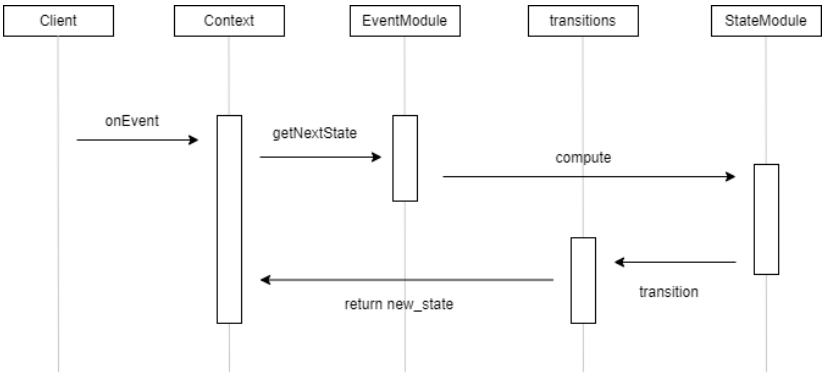

Figure 7. The invoking procedure of modular behavior model.

4. DESIGN OF OVERALL ARCHITECTURE

The overall architecture of the distributed cloud-based co-simulation CAE platform is high-performance and extensible and composable based on the fundamental module above. In the overall architecture of the CAE platform shows in Figure 8, we provide the support for data management and visualization, developer information security, CAE simulation construction and processing and so on.

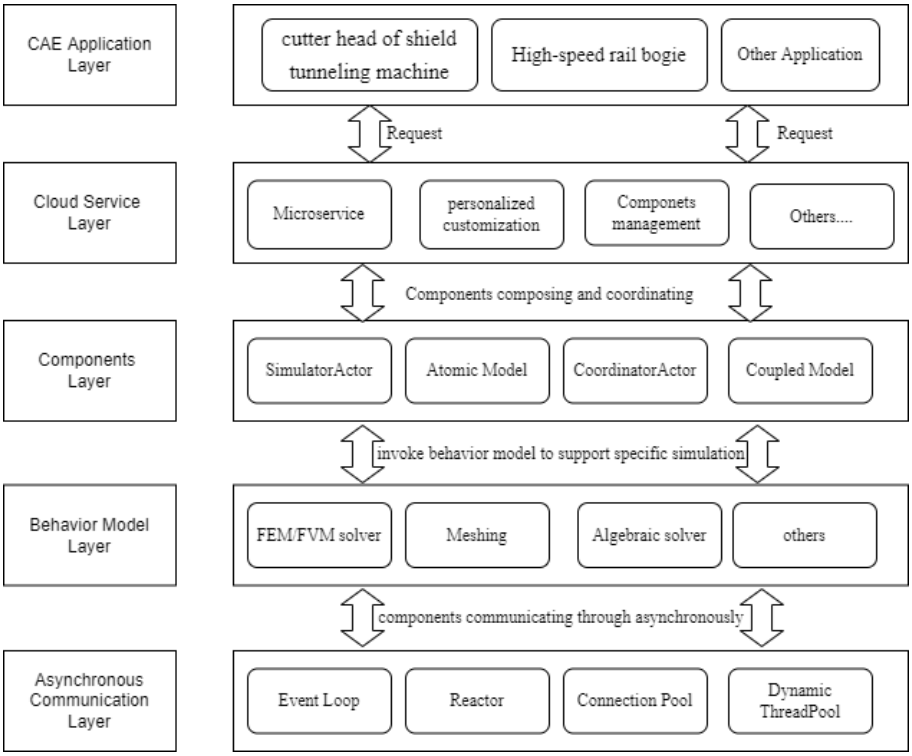

Figure 8.The overall architecture.

The CAE application layer is designed for users, it uses the Web browser as the client, and realizes the full-link CAE modeling and simulation through the graphical interface, including pre-processing, numerical solution, post-processing and other stages. And provide related engineering management, resource management, system management and other basic support functions. The cloud service layer calls various functions provided by the software integration layer by means of containerization, micro-service, efficient script driver, software component library, etc., to provide support for the requirements of the application layer. In the components layer, we implement the basic composable ActorDEVS architecture which incorporate Actor-oriented programming and M&S to support every stage in CAE simulation. Each

stage of CAE simulation will be designed into a coupled model or atomic model of DEVS, their relationships are described in the DEVS graphs. The data flows when the model states change. The behavior model layer is aimed to solve the issue of time-consuming tasks such as mesh, ODE/DAE solution, FEM/FVM solution and so on, they are the behavior model of the DEVS atomic model, users can choose specific solver when they start a new CAE task. Users can upload their implementations of any solver to the behavior model management functions which shows the extensibility and flexibility of our platform. In the communication layer, any network communication of the microservice and containers will using the fundamental asynchronous communication module.

5. CONCLUSION

The distributed cloud-based industrial software is key part of Industrial Internet and Industry 4.0, which solve the production and manufacturing issues through structuring, visualization, M&S and other means. In this paper, we propose two modules for distributed co-simulation engine. Firstly, we design an asynchronous communication module which implements high-performance data communicating and computing. Secondly, we design a composable distributed co-simulation module which adopts DEVS theory and Actor model. It enables automated coupling and composition of components and deployments in distributed scenario. On basis of module mentioned above, we implement a full-link cloud-based CAE platform which is very extensible and flexible.

ACKNOWLEDGEMENT

This work is funded by National Key R&D Program of China (No.2020YFB1709502). We would like to thank the corresponding authors: Jianwen Cao and Xuesong Wu for their invaluable contributions to this research project. Without their guidance and support, this work would not have been possible.

REFERENCES

[1] Mamad, Mohamed. (2018). Challenges and Benefits of Industry 4.0: an overview. International Journal of Supply and Operations Management. 5. 256-265. 10.22034/2018.3.7. Scheidegger A P G, Pereira T F, de Oliveira M L M, et al. An introductory guide for hybrid simulation modelers on the primary simulation methods in industrial engineering identified through a systematic review of the literature[J]. Computers and Industrial Engineering, 2018, 124: 474-492.

[2] Galvao Scheidegger, Anna Paula & Fernandes Pereira, Tábata & Oliveira, Mona Liza & Banerjee, Amarnath & Montevechi, José Arnaldo B.. (2018). An introductory guide for hybrid simulation modelers on the primary simulation methods in industrial engineering identified through a systematic review of the literature. Computers & Industrial Engineering. 124. 474-492. 10.1016/j.cie.2018.07.046.

[3] Peleg, David. (2000). Distributed Computing: A Locality-Sensitive Approach. 10.1137/1.9780898719772.

[4] Li, Shuya & Li, Qing & Zhang, Jianchao. (2023). MBSE-Based Turbofan Engine System Simulation Platform Design. 10.1007/978-981-19-6613-2_452.

[5] Zeigler, Bernard & Muzy, Alexandre & Kofman, Ernesto. (2018). Zeigler, B. P., Muzy, A., & Kofman, E. (2018). Theory of Modeling and Simulation: Discrete Event & Iterative System Computational Foundations. Academic Press.. 10.1016/C2016-0-03987-6..

[6] Wainer, Gabriel & Mosterman, Pieter. (2016). Discrete-Event Modeling and Simulation: Theory and Applications. 10.1201/9781315218731.

[7] Y. V. Tendeloo and H. Vangheluwe, "DISCRETE EVENT SYSTEM SPECIFICATION MODELING AND SIMULATION," 2018 Winter Simulation Conference (WSC), Gothenburg, Sweden, 2018, pp. 162-176, doi: 10.1109/WSC.2018.8632372.

[8] Shahin, Mojtaba & Ali Babar, Muhammad & Chauhan, Aufeef. (2020). Architectural Design Space for Modelling and Simulation as a Service: A Review. Journal of Systems and Software. 170. 10.1016/j.jss.2020.110752..

[9] Shuai, Li & Jing, Yang & Bin, Li & Bo, Li. (2011). Research based on CAD/CAE/CFD for collaborative design of automotive engines. 2. 1044 - 1047. 10.1109/ICMTMA.2011.542..

[10] Zhenhong, Han & Ju, Junxiu & Zhang, Yiyan. (2022). A web server optimization based on thread pool and epoll. 32. 10.1117/12.2639480.

[11] Lee, Kang-Lyul & Pham, Hong Nhat & Kim, Hee-seong & Youn, Hee & Song, Ohyoung. (2011). A Novel Predictive and Self -- Adaptive Dynamic Thread Pool Management. 93-98. 10.1109/ISPA.2011.61.
[12] Gatev, Radoslav. (2021). The Actor Model. 10.1007/978-1-4842-6998-5_10.
[13] Mailund, Thomas. (2022). Functional Programming. 10.1007/978-1-4842-8155-0_11.