Event-Based Data-Centric Semantics for Consistent Data Management in Microservices

Tilman Zuckmantel
University of Copenhagen
Copenhagen, Denmark
tizu@di.ku.dk

Yongluan Zhou
University of Copenhagen
Copenhagen, Denmark
zhou@di.ku.dk

Boris Düdder
University of Copenhagen
Copenhagen, Denmark
boris.d@di.ku.dk

Thomas Hildebrandt
University of Copenhagen
Copenhagen, Denmark
hilde@di.ku.dk

ABSTRACT

There is an emerging trend of migrating traditional service-oriented monolithic systems to the microservice architecture. However, this involves the separation of data previously contained in a single database into several databases tailored to specific domains. Developers are thus faced with a new challenge: features such as transaction processing, coordination, and consistency preservation, which were previously supported by the central database, must now be implemented in a decentralized, asynchronously communicating, distributed structure. Numerous prior studies show that these challenges are not met satisfactorily, resulting in inconsistent system states with potentially detrimental consequences. Therefore, we propose to design a coordination service that relies on clear event-based and data-centric formal semantics for microservices specifying the interaction of cross-microservice transactions with their respective databases. Furthermore, we provide a formalization of consistency properties and outline how they can be used to support dynamic monitoring as well as enforcement of consistency properties, thereby providing robust microservice systems. The envisioned architecture can significantly alleviate the developers' burden of implementing complicated distributed algorithms to maintain consistency across decentralized databases.

CCS CONCEPTS

• **Software and its engineering** → *Semantics*; • **Information systems** → **Distributed transaction monitors**.

ACM Reference Format:
Tilman Zuckmantel, Yongluan Zhou, Boris Düdder, and Thomas Hildebrandt. 2022. Event-Based Data-Centric Semantics for Consistent Data Management in Microservices. In *The 16th ACM International Conference on Distributed and Event-based Systems (DEBS '22), June 27–30, 2022, Copenhagen, Denmark.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3524860.3539807

1 INTRODUCTION

An increasing number of enterprises migrate their digital systems from the traditional monolithic architecture to the so-called microservice architecture, where components are implemented as independent and loosely-coupled applications. In doing so enterprises hope to benefit from the following advantages [19, 25]. Firstly, higher scalability can be achieved by a more fine-grained system with individual scaling potential. Secondly, robustness against server failures and errors can be increased by putting smaller responsibilities on each component. Furthermore, microservices can be maintained, tested and deployed individually. Finally, separating the central database into a database per microservice may lead to more maintainable data sets for each microservice.

However, in contrast to the benefits that companies expect from the microservice architecture, problems arise due to a lack of mechanisms to enforce data consistency. For example, ACID guarantees for transactions caused by the decentralized management of databases [16, 19, 22]. Often, transactions depend on data, not within their microservice, and thus need to access functionality from other microservices. We call these transactions cross-microservice transactions. Laigner et al. [19] in their work on current challenges of the microservice architecture outline that in order to control the exchange of data between microservices, application developers have to implement mechanisms themselves, resulting in redundant implementations and degrading data consistency guarantees. This paper focuses on two prevalent problems causing data consistency issues. First, missing transactional guarantees for interleaved cross-microservice transactions and, second, violation of event causality resulting in causally inconsistent data.

To close the gap and alleviate the burden on microservice developers, we propose introducing a scalable coordination service that controls the flow of events between microservices that interleave in cross-microservice transaction execution. To achieve this, we envision a novel approach to modeling microservice transactions semantically as automata, focusing on the interaction of the microservice with its respective database. We use the composition of automata to semantically express the interleaving of multiple cross-microservice transactions and thus create a foundation for providing transactional guarantees. Furthermore, we propose to explicitly observe causalities between events using formalized dynamic runtime monitors to prevent event causality violations. With this novel visionary architecture, microservice developers can be freed from implementing coordination algorithms and protocols to achieve consistent data management and instead focus on core business logic.

2 RELATED WORK

Orchestration is a popular method to coordinate cross-microservice transactions to achieve transactional guarantees, such as (a subset of) ACID properties [19–21, 26]. Several works have addressed the orchestration of distributed transactions. Traditional protocols such as 2PC can be used to achieve atomicity. However, to avoid

DEBS '22, June 27–30, 2022, Copenhagen, Denmark

ACM ISBN 978-1-4503-9308-9/22/06…$15.00
https://doi.org/10.1145/3524860.3539807

the blocking nature of 2PC, recently the SAGA pattern has become popular within the microservice community [20, 26]. The orchestrator-based SAGA pattern can provide atomicity guarantees without blocking participants by optimistically forwarding events and applying compensation actions for failed transactions. Both patterns lack to provide isolation guarantees. Some microservice development frameworks allow a centralized state store to achieve transaction isolation [12]. While this approach provides ACID guarantees, it violates the modularity of microservices, since generally decentralized data management is a desirable property for microservices [19]. Furthermore, Laigner et al. envision a centralized scalable data-store to achieve transactional guarantees [18]. However, their work lacks a formal semantic model to enforce transactional properties that we target in this paper. Recently, de Heus et al. have proposed a programming model for serverless functions that implements distributed transactions using two-phase commit and the SAGA pattern [10]. Nevertheless, their approach does not consider chaining multiple functions, which is necessary for microservices.

Moreover, to formally reason about properties of distributed systems, there are several semantic approaches. Multi-Party Session Types have been introduced to formally verify communication protocols between multiple participants [14]. While session types can guarantee order and type correctness of messages, they don't provide a way to reason about the state change of data. Recently, Jangda et al. proposed a semantic model for serverless functions [15]. By expressing the behavior of serverless functions as operational semantics, the authors provide a sound implementation that also reasons about the interaction with a centralized key-value store. Microservices, however, require more components to be semantically expressed than serverless functions. In particular, a semantic model of microservices must be able to capture an individual database per microservice, a prevalent microservice design pattern. Furthermore, Ouederni has recently shown a way to model asynchronously communicating distributed components using interface automata and linear transition systems [23]. Following Ouedernies approach it is possible to formally verify properties such as deadlock freedom and unexpected recipients, but it is not intended to model interactions with decentralized databases.

3 PROBLEMATIC SCENARIOS

The basis for the presented problematic scenarios is a microservice architecture from the domain of e-commerce, consisting of three microservices. One microservice is responsible for managing orders (`Orders`), another for managing payment processes (`Payments`), and a final one for managing customers (`Customers`). Each microservice consists of several components. Firstly, an individual key-value store. Secondly, an API that represents the business logic of the individual microservice, and finally, an event queue that delegates events to the corresponding API functions. Consequently, communication in this architecture is event-based and asynchronous. While the literature highlights many challenges in system development with microservices, we, in this paper, focus on the following prevalent data consistency problems [8, 19, 24].

(1.) Transactional Guarantees: Figures 1a and 1b show two cross-microservice transactions from the `Orders` and the `Customers` microservice to place an order for a given customer and delete a customer, respectively. Figure 1c shows an interleaving of the

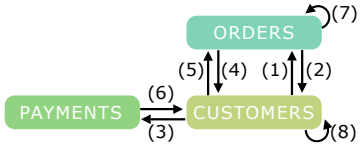
```
placeOrder(id,order):
 //req:(4), resp:(5)
let c =
 existsCustomer(id);

if(c){
  //insertion:(7)
  db.insert(id,order);
  send(PlaceOrdSucc);}
else{send(PlaceOrdFail);}
```

(a) Cross-microservice transaction for placing an order for a customer if the customer exists.

```
deleteCustomer(id):
//req:(1), resp:(2)
let o = existsOrdFor(id);
//req:(3), reps:(6)
let p = existsPayFor(id);
if(!(o && p)){
  //deletion:(8)
  db.delete(id);
  send(DeleteSucc);}
else{send(DeleteFail);}
```

(b) Cross-microservice transaction for deleting a customer if there are no proceeding orders or payments.



(c) Interleaving of functions illustrated in figure 1a and 1b. Numbers in the comments of the source code represent numbers in the interleaving and determine the global order of exchanged events.

Figure 1: Implementation of two cross-microservice transactions in two different microservices resulting in a globally inconsistent state when interleaved as shown in figure 1c.

transactions in which `deleteCustomer` starts by checking if there are existing orders for the customer (1-2). It further proceeds by checking for any uncompleted payments (3). While the `Payments` microservice checks upon open payments, the `placeOrder` transaction is triggered in the `Orders` microservice targeting the same customer. It starts by checking upon the existence of the customer (4-5), and because the `deleteCustomer` transaction is pending, the customer record still exists. Thus, the returned value will evaluate to `true`. Subsequently, the `Payments` microservice answers with the result of existing payments (6). Finally, the customer is deleted from the database while, at the same time, a new order is placed (7-8). This exemplifies an uncontrolled data race between the two microservices leading to an inconsistent global state caused by the decentralized management of databases. While a centralized database in a conventional monolithic architecture ensures transaction guarantees, it is the task of the application developer to prevent race conditions in microservices. Possible methods to control cross-microservice transactions include distributed locking mechanisms, distributed commit protocols such as 2PC, and orchestrator-based techniques such as the orchestrator-based SAGA pattern [19, 20, 22]. However, using distributed locking and 2PC implies coupling of transactions, and the SAGA pattern can not guarantee isolation of data records. To control the scenario shown here, we are looking for a solution based on orchestration that additionally provides isolation guarantees for the specified transactions.

(2.) Causal Consistency: Exemplary consider a scenario in which a customer requests reimbursement for a product that has been ordered. However, due to internal processing details of the `Payments` microservice, the payment for this order has not yet been proceeded successfully. Logically, the system should not allow an event that represents the request for reimbursement to process until a successful payment event has been observed. This connection between

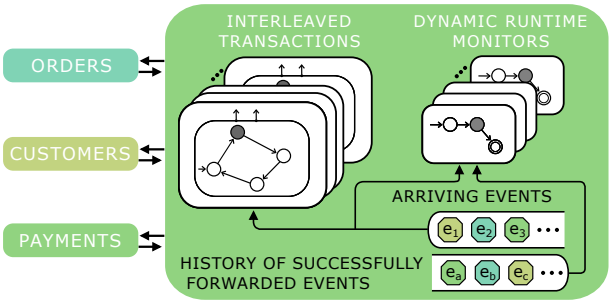


Figure 2: Visionary Architecture connecting microservices through a coordinator.

the two events can be interpreted as a happens-before explicit event causality that implies an order upon the two events and ultimately an order between the operations responsible for proceeding with a payment and a reimbursement [1, 17]. However, because functionalities are distributed in many different microservices and microservices are developed independently from each other, it is easy to overlook such causalities between events. Consequently, a solution should support the ability to globally establish such causalities and alert system administrators of violations or even prevent violations.

4 VISIONARY ARCHITECTURE

Conventional concurrency control mechanisms can not be readily applied to achieve transactional guarantees for cross-microservice transactions because the data operations scattered across multiple microservices are typically not externally observable. Therefore, we need a mechanism to capture cross-microservice transactions semantically so that it is possible to globally observe the data operations of multiple microservice transactions and reason about their consistency properties. To control the flow of interleaved transactions, we need a way to influence the order in which events are sent between microservices. This way, we can predict and control which events, when sent, will lead to safe transaction steps in the affected microservices. To guarantee atomicity, we need a mechanism that can revise the effect on microservice databases of an aborted cross-microservice transaction. Furthermore, to prevent duplicated events, the mechanism must be able to filter events that are not expected. Finally, to satisfy explicit causality, we propose that a solution can evaluate temporal causality constraints beyond the lifetime of one or more interleaved cross-microservice transactions based on the history of successfully processed events.

Frameworks that attempt to address cross-microservice transactions challenges typically resort to centralized components. The Axon framework, for example, uses a centralized event-bus component for coordination, while Dapr relies on a state store across multiple microservices to achieve cross-microservice transaction guarantees [7, 12]. Furthermore, the popularity of the SAGA pattern is growing within the microservice community [19, 20, 26], which provides a way to control the execution of distributed transaction operations through a central orchestrator or choreographies to achieve a certain degree of atomicity. Our proposal can be seen as a novel extension of these approaches with formal semantics for cross-microservice transactions to solve the aforementioned data consistency problems with a coordinator.

More specifically, we envision a coordinator depicted in figure 2 controlling the order of events respective to the states of microservices. Microservice functions are seen as state transformers, triggered by received events and producing responding events. The state is reflected in the database, and the state transitioning can be expressed in the form of an extended interface automaton [9]. Each transition in an automaton represents one or more steps in the application logic of the microservice function, with CRUD operations being directly represented. In addition to the application logic of a microservice function, interface automata specify the input and output of messages. This allows capturing the outgoing and incoming events during the execution time of a transaction. Multiple interface automata representing microservice functions can be composed over the exchange of events [9]. This provides formal semantics of concurrent transaction execution, focusing on the data interactions and the event exchange of each microservice.

Microservices that want to initiate operations in other microservices send events to the central coordinator. The central coordinator can determine, given the current state of the interface automaton representing the interleaved transactions, whether a state transition does not compromise transaction guarantees. We can use interface automaton as a look-ahead of database operations to observe future database operations on event forwarding. This also includes the prevention of forwarding duplicated events during a transaction, since an event that arrives at the central coordinator unexpectedly will not be forwarded. In addition, we propose to compose as interface automaton only transactions that endanger transaction guarantees in an interleaving. Transactions whose database operations are non-conflicting, for example, can be considered separately. Following, the coordinator controls multiple instances of interface automaton representing interleaved transactions in execution. After completing interleaved transactions, the corresponding interface automaton is completed and removed from the coordinator.

We use interface automaton to safeguard transaction guarantees of interleaved cross-microservice transactions. However, because automata complete on transaction completion, it is impossible to observe explicit causality constraints beyond the lifetime of concurrent transactions. Therefore, we propose to use dynamic runtime monitors that reason about explicit causality constraints given the history of successfully forwarded events respective to arriving events. There are four main criteria for selecting an approach to specify and monitor runtime properties: (1) It must be possible to express relevant properties, such as causal consistency. In particular, it should be possible to express both safety and liveness properties and refer to the timing and attributes of events. (2) It should be possible for the engineer to specify these properties or to derive them from the code. (3) There must exist effective monitors for the chosen specification language. (4) There should be techniques for preventing violations when possible.

To meet these criteria, we propose to derive a specification language by taking the best from Metric First-Order Temporal Logic (MFOTL) [3] and the timed Dynamic Condition Response (DCR) graph formalism [2]. In [3], Basin et al. use MFOTL to monitor safety policies similar to the causality properties of microservices. What makes MFOTL particularly relevant for microservices is that it allows for both effective monitoring and the specification of timing and causal properties of events and quantification over values for

attributes of events, e.g., the transaction identifiers as exemplified in the next section. In [2] on the other hand, Basin et al. show how to apply the formalism of timed DCR graphs to specify, monitor, and pro-actively timed safety properties. Compared to the work based on MFOTL, the work in [2] does not allow for quantification over values but provides an algorithm for proactively preventing violations of properties by the use of controllable events of the monitored systems, similar to what is found in supervisory control systems [5]. Recent work on timed DCR graphs has extended the formalism to allow for the specification and computations of values [13] and replication of sub processes [11], which indicates a possible way for the combination of the use of the techniques in [3] and [2] to provide adequate monitoring and pro-active prevention of property violations of timed microservices.

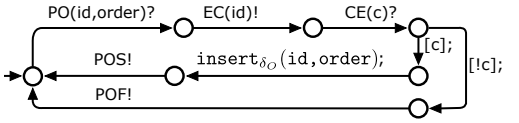
Since we can observe the exact semantics of database operations performed in individual microservices in the central coordinator, we can also execute them here. This allows us to take advantage of the fact that data relevant to the run-time monitors is available in the central coordinator consistently with the data from the microservice without further coordination.

Last, our approach provides a methodology that falls between orchestration, typically executing business logic in the orchestrator, and choreography-based techniques, which are decentrally executed. Thus, the developer is faced with a trade-off between modularity and safety: critical transactions that require strict consistency use the coordinator. If eventual consistency is sufficient, events do not need to pass the coordinator.

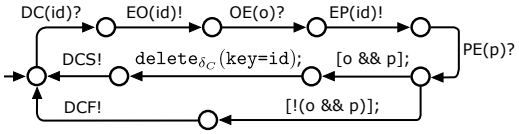
Summarizing the envisioned mechanism solves the problems stated in the latter section. Using a coordinator following the data-centric semantics of cross-microservice transactions guarantees consistency and isolation. The observability of transaction progress makes it possible to reverse one or more transactions if necessary. Last, constraints can be added as run-time monitors to ensure explicit causality to a degree specified by the developer.

5 EVENT-BASED DATA-CENTRIC SEMANTICS

Cross-microservice Transaction Semantics: We envision the usage of interface automaton to capture the semantics of cross-microservice transactions [9]. Interface automata define a specification of a component by its communication behavior using input and output messages (denoted by `?` and `!` respectively) with additional internal actions (denoted by `;`) to transition between states. While traditional interface automaton supports message exchange, we need more fine-grained control over the content of events to semantically base decisions on event payloads. Furthermore, a mechanism is needed to make database interactions observable within an interface automaton to capture the database's state explicitly. Therefore, we propose to add three novel elements to the traditional interface automaton model presented in [9]. Figure 3 shows our envisioned interface automata that semantically represent the cross-microservice transactions in figure 1. (1.) We add the possibility to explicitly add event payloads to messages as attributes of the event. This is exemplarily illustrated by the `PO(id,order)?` transition semantically expressing the receiving of an event in figure 3a. Here, `id` and `order` are attributes of the event and used in further automaton transitions. (2.) We add branching semantics over boolean values to support branching upon the payload elements of events used, for



(a) Interface Automaton semantically expressing the transaction in figure 1a. Abbreviations: PO:PlaceOrder, EC:ExistsCustomer, CE:CustomerExists, POS:PlaceOrderSuccess, POF:PlaceOrderFailure.



(b) Interface Automaton semantically expressing the transaction in figure 1b. Abbreviations: DC:DeleteCustomer, EO:ExistsOrder, OE:OrderExists, EP:ExistsPayment, PE:PaymentExists, DCS:DeleteCustomerSuccess, DCF:DeleteCustomerFail.

Figure 3: Visionary interface automata for cross-microservice transaction semantics.

example, in the interface automaton in figure 3a to decide further steps in the automaton based on the value of `c`. (3.) We add CRUD operations as internal actions to the interface automaton. A CRUD operation transition defines the type of its action and the database over which it operates. Transition labelled $\texttt{insert}_{\delta_O}\texttt{(id,order)}$ in the interface automaton in figure 3a for example inserts an order for a key with value `id` into key-value store δ_O of the orders microservice.

Multiple interface automata can be composed into a new component according to their communication exchange. Accordingly, we propose to use the composition of transactions for an entire semantic representation of a cross-microservice transaction. Composing the two interface automaton in figures 3a and 3b together with the cross-microservice requests contained in the respective transactions yields a new interface product automaton semantically expressing all paths that the two transactions can interleave with synchronous communication. While the semantic model does not capture asynchronous communication, it does not prevent microservices from communicating asynchronously because the coordinator does not need to know the exact time of events being sent or received to maintain the automaton state transition. If both transactions start concurrently, the coordinator will construct the composition, process expected events from both microservices, and update the state respectively. Recap the concurrency problem in the interleaving in figure 1c. To prevent the problem that arises from the uncontrolled interleaving shown here, we can use the coordinator to create the composition of the executed transactions and control the flow of events between the microservices. Figure 4 shows a relevant excerpt of this composition to show how a violation of serializability can be proactively prevented. Here, the path executed by the ordering in Figure 1c in the composition of the aforementioned transactions is shown. Note that the $\texttt{read}_{\delta_C}\texttt{(id)}$ transition represents the lookup of a customer inside the database of the customer microservice and is included in the composition because of the cross-microservice request in figure 1a that checks for the existence of a customer. The patterned state of the composition indicates the state of the interleaving inside the coordinator. In the next step of the problematic interleaving, a `PaymentExists` event addressed from the

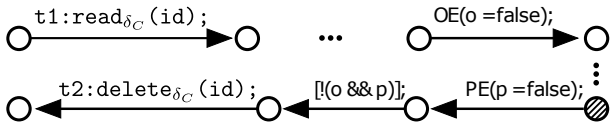


Figure 4: Path in an interface automaton representing the interleaving in figure 1c.

payment microservice to the customer microservice would pass the coordinator. However, given the semantics of the future steps executed inside the customer microservice, it is possible to recognize in advance that forwarding this event will cause a read-write conflict based on the history of transition steps. Here, the transition `read`$_{\delta_C}$`(id)` of transaction `t1` would conflict with the transition `delete`$_{\delta_C}$`(id)` of transaction `t2`.

Furthermore, because we forward events optimistically, we can reach a state where no further transitions are safe. To abort a transaction, we can extract and compose the operations necessary to reverse the steps of a transaction from the semantic model. This is possible because we can trace the path that led to the current state in the interface automaton and compose information about executed semantic steps. The composed steps can then be sent to the corresponding microservices. It is the individual microservices' responsibility to process these compensation operations.

Distinction from lock managers: It can be conceivable to use a lock manager to guarantee isolation, but we would need a semantic representation of microservice transactions to automatically infer locks centrally to avoid requiring the application developer to manually manage locks. The observable event flow is insufficient to infer triggered database operations inside microservices. In contrast, interface automata can prevent deadlocks proactively by using the semantics for look-ahead of database operations. Additionally, interface automata can be used to extract semantic information needed to abort transactions safely.

Explicit Event Causality Semantics: To observe explicit event causalities like the reimbursement example from section 3, interface automata that semantically express interleaving cross-microservice transactions are insufficient. The reason is that these causalities need to be observed beyond the lifetime of multiple interleaved transactions. Therefore, we envision introducing formalized event constraints defined over the history of successfully forwarded events that can be translated to dynamic runtime monitors. It is possible to express explicit event causalities with the help of temporal logics such as the Metric First Order Temporal Logic (MFOTL) [3]. MFOTL allows to define predicates and quantify over an infinite domain of individuals, making it a suitable candidate to operate over a history of events. The following formula shows a formalized event constraint expressing that for every reimbursement for a specific *id* there must have been a successful payment in the past.

$$\Box \forall id.\text{reimbursement}(id) \Rightarrow \blacklozenge \text{paySuccess}(id)$$

Using the results of Basin et al. in [3, 4] it is possible to monitor such temporal formulas efficiently over a stream of `Reimbursement` and `PaySuccess` events. By combining this approach with the later work of Basin et al. on proactive enforcement [2] based on timed DCR graphs, we can also achieve proactive prevention of violations. Concretely, the following timed DCR-graph expresses that if a `Reimbursement` event happens without a preceding `PaySuccess`, a `ReimbursementFailed` event must happen within 60 minutes in the future:

$$\begin{aligned} \text{Reimbursement} \overset{60}{\bullet\!\!\rightarrow} \text{ReimbursementFailed} \mid \\ \text{PaySuccess} \rightarrow\!\!\% \text{ ReimbursementFailed} \mid \\ \text{Reimbursement} \rightarrow\!\!\% \text{ PaySuccess} \end{aligned}$$

The policy can be enforced using the approach in [2] by making the `ReimbursementFailed` action causable, meaning that the monitor can cause the activity to happen. The causality constraints are defined based on events. Therefore, we do not expose the internal data of microservices, preserving the encapsulation principle. However, the event schema must be globally known.

Complexity: Composition of n interface automata is exponential with respect to n. However, we believe composition can be efficiently accomplished in the implementation for real-world microservice systems. First of all, composition of n components does not necessarily lead to an exponential size requirement because the composition operation expresses synchronous communication. This means that instead of covering all possible paths of asynchronous communication interleaving, it only captures the synchronous path, thereby significantly reducing the complexity of the product automaton. Furthermore, interface automata are not input-enabled. This means that it is not required to have a transition for each element of the input alphabet in every state. Following this leads to fewer transitions in the product automaton.

Second, to monitor cross-microservice transactions, it is not necessary to construct the composition of two automata. Since the composition simulates the interleaved execution, it is also possible to maintain the state in each automaton individually. Whether an arriving event can perform a state transition must then be decided based on all interface automata involved in the transaction.

6 OPPORTUNITIES AND CHALLENGES

Several challenges have to be addressed to realize the aforementioned visionary solution. The first open question is how to generate interface automata from source code in a microservice. To minimize the work of the application developer, we ideally seek a mechanism that automatically extracts as much semantic information about the source code as possible. Information about data types, references within a microservice, and branching via if-conditions can be extracted by constructing an abstract syntax tree during the compilation step of the source code. Other information must be added as follows and is illustrated in figure 5.

(1) Identify cross-microservice transactions to be semantically expressed as interface automaton.`@CrossMicroserviceTransaction` annotation indicates that this business function should be translated into an interface automaton, adds a globally unique identifier `PO` for global dependency resolution and specifies which event triggers the functions. (2) Resolve cross-microservice dependencies. Here, inheritance with a generic base class is used to represent cross-microservice API requests. Overriding the `send` function allows for implementing the event forwarding framework-independent while enriching source code with necessary semantic information. (3) Identifying the database relevant for the interface automaton. We envision a wrapper class for the database that requires the implementation of four CRUD operations. Furthermore, to state

```
class ExistsCustomerRequest extends
  CrossMicroserviceAPIRequest<ExistsCustomer,CustomerExists> {
  send(e:ExistsCustomer):CustomerExists = {...}}

@CrossMicroserviceTransaction(id=PO,input=PlaceOrder)
placeOrder(id,order):
 let db = new DatabaseWrapper(...)
 let c:Boolean = new ExistsCustomerRequest()
   .send(new ExistsCustomer(id)).then(res => res.payload())
 if(c){
   db.insert(id,order); reply(PlaceOrdSucc);
 } else {reply(PlaceOrdFail);}
```

Figure 5: Envisioned enrichment of the transaction in Fig. 1a to generate an interface automaton from source code.

causality constraints conveniently, we envision the usage of a SQL-like Domain Specific Language (DSL) for stating invariants that are then translated to formal monitors. The DSL expresses causalities between events supporting commonly needed invariant patterns as demonstrated in [19]. Its expressivity is bound by the underlying formal model of MFOTL [3].

For an efficient implementation, we propose to extend an existing framework such as Dapr [12] with the global coordinator as an additional component. We argue that a global coordinator would not limit the throughput of the system. Essentially, the coordinator's tasks can be considered as stream processing jobs. We envision that techniques of scalable stream processing systems [6] can be applied to scale the coordinator horizontally. Conflict-free transactions, for example, do not need to be maintained inside the same instance of the coordinator. In addition, dynamic runtime monitors that do not interfere with their observed event types can also be considered in separate instances.

The visionary semantic elements presented in this paper to solve the three problems from section 3 do not cover all existing problems. For example, it is helpful to consider data constraints like foreign key constraints when deleting data or introducing data invariants to determine valid data ranges.

7 CONCLUSION

In this work, we have considered three prevalent problems related to the consistency challenges of decentralized database management in current microservice architectures. We propose to capture the behavior of cross-microservice transactions using automata, dynamic runtime monitors, and proactive controllers in a coordinator to secure causality relationships between events. We have also identified several research challenges and opportunities to realize the envisioned solution. Using this paper's approach, we believe we can support more robust microservice systems with stronger consistency guarantees, removing the burden of implementing consistency measures from the application developer.

ACKNOWLEDGMENTS

Part of the PAPRICAS.org project supported by Independent Research Fund Denmark (DFF.dk).

REFERENCES

[1] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2012. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the Third ACM Symposium on Cloud Computing - SoCC '12*. ACM Press, San Jose, California, 1–7.

[2] David Basin, Søren Debois, and Thomas Hildebrandt. 2016. In the Nick of Time: Proactive Prevention of Obligation Violations. *I E E E Computer Security Foundations Symposium. Proceedings*, 120–134. IEEE Computer Security Foundations Symposium, CSF ; Conference date: 27-06-2016 Through 01-07-2016.

[3] David Basin, Felix Klaedtke, and Samuel Müller. 2010. Monitoring security policies with metric first-order temporal logic. In *Proceeding of the 15th ACM symposium on Access control models and technologies - SACMAT '10*. ACM Press, Pittsburgh, Pennsylvania, USA, 23.

[4] David Basin, Felix Klaedtke, Samuel Müller, and Eugen Zălinescu. 2015. Monitoring Metric First-Order Temporal Properties. *J. ACM* 62, 2 (May 2015), 1–45.

[5] Bertil Brandin and Walter Wonham. 1994. Supervisory control of timed discrete-event systems. *Automatic Control, IEEE Transactions on* 39 (03 1994), 329 – 342.

[6] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38 (2015), 28–38.

[7] Binildas Christudas. 2019. Axon Microservices and BASE Transactions. In *Practical Microservices Architectural Patterns*. Apress, Berkeley, CA, 779–812.

[8] Binildas Christudas. 2019. Transactions and Microservices. In *Practical Microservices Architectural Patterns*. Apress, Berkeley, CA, 483–541.

[9] Luca de Alfaro and Thomas A. Henzinger. 2001. Interface automata. *ACM SIGSOFT Software Engineering Notes* 26, 5 (Sept. 2001), 109–120.

[10] Martijn de Heus, Kyriakos Psarakis, Marios Fragkoulis, and Asterios Katsifodimos. 2021. Distributed transactions on serverless stateful functions. In *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems*. ACM, Virtual Event Italy, 31–42.

[11] S. Debois, T.T. Hildebrandt, and T. Slaats. 2018. Replication, refinement & reachability: complexity in dynamic condition-response graphs. *Acta Informatica* 55 (2018), 489–520.

[12] Radoslav Gatev. 2021. State Management. In *Introducing Distributed Application Runtime (Dapr)*. Apress, Berkeley, CA, 145–158.

[13] Thomas T. Hildebrandt, Håkon Normann, Morten Marquard, Søren Debois, and Tijs Slaats. 2022. Decision Modelling in Timed Dynamic Condition Response Graphs with Data. In *Business Process Management Workshops*, Andrea Marrella and Barbara Weber (Eds.). Springer International Publishing, Cham, 362–374.

[14] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* 63, 1 (March 2016), 1–67.

[15] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. 2019. Formal foundations of serverless computing. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), 1–26.

[16] Miika Kalske, Niko Mäkitalo, and Tommi Mikkonen. 2018. Challenges When Moving from Monolith to Microservice Architecture. In *Current Trends in Web Engineering*, Irene Garrigós and Manuel Wimmer (Eds.). LNCS, Vol. 10544. Springer International Publishing, Cham, 32–47.

[17] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. 1992. Providing high availability using lazy replication. *ACM Transactions on Computer Systems* 10, 4 (Nov. 1992), 360–391.

[18] Rodrigo Laigner, Yongluan Zhou, and Marcos Antonio Vaz Salles. 2021. A distributed database system for event-based microservices. In *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems*. ACM, Virtual Event Italy, 25–30.

[19] Rodrigo Laigner, Yongluan Zhou, Marcos Antonio Vaz Salles, Yijian Liu, and Marcos Kalinowski. 2021. Data management in microservices: state of the practice, challenges, and research directions. *Proceedings of the VLDB Endowment* 14, 13 (Sept. 2021), 3348–3361.

[20] Konstantin Malyuga, Olga Perl, Alexandr Slapoguzov, and Ivan Perl. 2020. Fault Tolerant Central Saga Orchestrator in RESTful Architecture. In *2020 26th Conference of Open Innovations Association (FRUCT)*. IEEE, Yaroslavl, Russia, 278–283.

[21] Davi Monteiro, Paulo Henrique M. Maia, Lincoln S. Rocha, and Nabor C. Mendonça. 2020. Building orchestrated microservice systems using declarative business processes. *Service Oriented Computing and Applications* 14, 4 (Dec. 2020), 243–268.

[22] Raja Mubashir Munaf, Jawwad Ahmed, Faraz Khakwani, and Tauseef Rana. 2019. Microservices Architecture: Challenges and Proposed Conceptual Design. In *2019 International Conference on Communication Technologies (ComTech)*. IEEE, Rawalpindi, Pakistan, 82–87.

[23] Meriem Ouederni. 2021. Compatibility checking for asynchronously communicating software. *Science of Computer Programming* 205 (May 2021), 102569.

[24] Christian Posta. 2016. The Hardest Part About Microservices: Your Data. https://blog.christianposta.com/microservices/the-hardest-part-about-microservices-data/

[25] IBM Team. 2021. *Microservices in the enterprise, 2021: Real benefits, worth the challenges*. Technical Report. International Business Machines Corporation.

[26] Martin Štefanko, Ondřej Chaloupka, and Bruno Rossi. 2019. The Saga Pattern in a Reactive Microservices Environment:. In *Proceedings of the 14th International Conference on Software Technologies*. SCITEPRESS - Science and Technology Publications, Prague, Czech Republic, 483–490.