IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 29 November 2023, accepted 22 January 2024, date of publication 1 February 2024, date of current version 8 February 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3361318

RESEARCH ARTICLE

Transaction Conflict Control in Hyperledger Fabric: A Taxonomy, Gaps, and Design for Conflict Prevention

MÁTÉ DEBRECZENI, ATTILA KLENIK, AND IMRE KOCSIS

Department of Measurement and Information Systems, Budapest University of Technology and Economics, 1117 Budapest, Hungary

Corresponding author: Imre Kocsis (kocsis.imre@vik.bme.hu)

This work was supported in part by the Cooperation Agreement between the Hungarian National Bank (MNB) and the Budapest University of Technology and Economics (BME) in the Digitization, Artificial Intelligence, and Data Age Workgroup. The work of Attila Klenik and Imre Kocsis was supported in part by the SME4DD Project of the European Union's Digital Europe Program under Project 101100768.

ABSTRACT The execute-order-validate approach to blockchain consensus, most notably implemented by Hyperledger Fabric, facilitates highly scalable execution of smart contract – in Fabric terminology, "chaincode" – invocations in cross-organizational blockchains; at the expense of requiring multi-version concurrency control conflict handling during block validation. Consequently, the system-level goodput can be significantly lower than throughput. Although several solutions have been proposed for handling and avoiding conflicts in Hyperledger Fabric, a systematic and holistic approach is missing. We introduce the notion of conflict-controlled operation, propose a novel taxonomy of its means based on the codified principles of dependable computing, and categorize the known approaches. Based on this taxonomy, we identified the critical gaps in the state-of-the-art. Design-time conflict prevention is one such gap, and we propose the application of a model-driven engineering process for this purpose. For the last storage mapping stage of the process, we propose entity attribute partitioning for conflict prevention, describe a data mapper-style chaincode layer, and empirically evaluate our solution.

INDEX TERMS Blockchain, hyperledger fabric, multi-version concurrency control, goodput, taxonomy, dependable computing, model-driven engineering, attribute affinity.

I. INTRODUCTION

Blockchain-based business solutions and applications began to proliferate in the past decade since the inception of the *smart contract*-enabled Ethereum [1] technology and network. Today, numerous enterprise-grade platforms facilitate the creation and management of *closed access* and *permissioned consensus* [2] blockchains, serving specific cross-organizational collaboration use cases in a dedicated way. Hyperledger Fabric (HLF) [3] is one of the most popular and mature closed-permissioned blockchain platforms.

Fabric offers a modular, performant, and scalable solution to satisfy the diverse requirements of enterprise use cases. It achieves new levels of scalability (at least in the context of blockchains) through a consensus protocol that employs *multi-version concurrency control* (MVCC; a form of optimistic concurrency control), allowing more parallel access to the shared blockchain data than blockchains architected for the open-unpermissioned setting.

While traditional database solutions have been using MVCC for a long time [4], Fabric introduced it for blockchains by applying a novel *execute-order-validate* consensus approach – instead of the well-known *order-execute* paradigm, notably followed by Ethereum.

However, the application of MVCC makes transaction success highly dependent on the manner in which smart contracts read and write data. Unfortunately, designed data models and smart contracts can lead to many failed transactions (owing to MVCC conflicts), significantly limiting the system's *goodput* and *scalability*. Blockchain clients must also re-submit such

The associate editor coordinating the review of this manuscript and approving it for publication was Barbara Guidi.

© 2024 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

transactions, resulting in an additional load on the system, which potentially negatively affects the execution of other transactions in the performance domain. Finally, transactions that fail because of MVCC conflicts also represent wasted resources because the transaction failure is unsubstantiated at the business logic level. Although several solutions have been proposed for handling and avoiding MVCC conflicts in Hyperledger Fabric, a systematic and holistic approach supporting requirement-based design is still missing.

In this paper, we make the following contributions.

- We introduce the notion of *conflict-controlled operation* for Hyperledger Fabric.
- Building on a core taxonomy of dependable computing, which is known to support extra-functional assurance design in system engineering processes, we propose a taxonomy of the means of conflict-controlled operation.
- We survey and categorize the known MVCC conflict-addressing approaches and identify the gaps in the state-of-the-art.
- We propose the application of the concepts and methods of model-driven engineering for planned conflict prevention.
- We propose two specific approaches to conflict prevention in mapping platform-independent concepts to the key-value storage model of Fabric. We describe a supporting SDK prototype and provide an empirical evaluation.

The paper is structured as follows. Section II summarizes the consensus mechanism of Hyperledger Fabric and the ways MVCC conflicts emerge. Section III introduces our taxonomy, categorizes the known approaches, and identifies important gaps. Section IV describes the common model refinement methodology of model-driven engineering and argues that it is able to support the design of conflict prevention by facilitating the proper mapping of the instances of business-level concepts to the key-value pairs stored by Hyperledger Fabric. We evaluate the state-of-the-art of model-driven engineering and low-code/no-code development for Fabric from the point of view of recognizing the need to design against MVCC conflicts. Section V proposes conflict prevention with storage-level entity attribute partitioning, presents "total partitioning" and an attribute affinity-based partitioning algorithm, describes our prototype chaincode SDK, and provides initial empirical evaluation. The code, data, and analysis artifacts associated with this section are available in our GitHub repository.[1]

The paper is based on the initial results of a student research report [5], created by one of the authors (and consulted by two others). However, only Section V of this paper relies on the report meaningfully, and that content has also been significantly revised.

II. EXECUTE-ORDER-VALIDATE CONSENSUS IN FABRIC

This section gives an overview of the relevant core concepts of Hyperledger Fabric and its consensus protocol. For further details, we refer to [3] and the Fabric documentation.[2]

A. TRANSACTION PROCESSING

A Hyperledger Fabric network is operated by a membership-controlled *consortium* of organizations with the goal of jointly maintaining a connected set of distributed key-value ledgers, each equipped with smart contracts (in Hyperledger Fabric terminology, the ledgers are called *channels* and smart contracts *chaincode*). The ledgers do not have a native data model (notably, there is no "unit of value", or cryptocurrency, present out of the box); ledger content is entirely defined by the key-value operations performed by the smart contracts.

Each channel is maintained by a subset of the organizations, as determined by business cooperation requirements and the intended balance between integrity and need-to-know considerations. Without losing generality, throughout this paper, we assume a single-channel Fabric network, as cross-channel chaincode operations are restricted to reads. As we will see, this means that even in a multi-channel setting, for each transaction, only the channel it directly targets will be relevant from the point of view of MVCC conflicts.

The distributed ledgers realized by a Hyperledger Fabric network are blockchain-based; however, the Fabric system architecture and consensus approach are significantly different from public blockchains. In Fabric, consensus and client access are both permissioned – the latter is restricted to members and systems of the participating organizations; it is common to call Fabric networks *consortial blockchains*.

Each organization contributes resources to the network in a number of roles. *Peer* nodes are the primary computational resources of the network, responsible for maintaining the distributed ledger by executing, validating, and committing transactions.

A set of *ordering service nodes* (OSNs) jointly operates the *ordering service*, which determines the global order of transactions and batches them to blocks. Organizational *clients* submit transaction requests to the network to trigger ledger state changes subject to authentication, authorization, and the Fabric consensus protocol (depicted in Figure 1).

1) The client sends a *transaction proposal* to peers of the participating organizations, containing various *parameters*, such as the *smart contract* (*chaincode*) function to execute and its inputs.
2) Peers independently execute the proposals against their stored distributed ledger state by executing the target *chaincode*, a *smart contract* that encodes the necessary business logic. Chaincodes access the current peer-local ledger state (the *world state*) through the executor peer as a *versioned key-value store*. The peer records the data accesses of the executing transaction

[1] https://github.com/ftsrg/hyperledger-fabric-mvcc-analysis

[2] https://hyperledger-fabric.readthedocs.io/en/latest/

in the form of a *read set* and a *write set*, but does not perform the writes.

3) The client receives the execution – "simulation" – results from the peers in terms of versioned variable reads and writes. Each peer *endorses* and signs successful executions in the name of its controlling *organization*. The client receives the read set and the write set as part of an endorsement.
4) If the client gathered sufficient endorsements with matching read sets and write sets, it submits the proposal, endorsements, and read and write sets to an OSN for *ordering*. The number of necessary endorsements is governed by a configurable *endorsement policy* (e.g., one endorsement from at least k participating organization out of n).
5) The OSNs use the crash fault-tolerant Raft protocol [6] to *globally order* the incoming transactions of clients in a first-in, first-out manner. (The ordering service is pluggable – other options exist – but Raft is the preferred one.)
6) When a preconfigured criterion is met (number of transactions, aggregated payload size, or timeout), the ordering service creates and *broadcasts* a new block of transactions to the peers.
7) The peers receive the new block and *validate* multiple aspects of the transactions in the block. The MVCC validation substep ensures that the read set of each transaction still matches the current world state (see below). Only the write sets of valid transactions are *committed* to the world state.
8) Finally, the peers *notify* the subscribed clients about the final statuses of new transactions in the latest block.

Peers in newer versions of Fabric can also provide a *gateway* functionality, performing most of the client's work: steps ① through ④ and ⑧ in Figure 1 are pushed into the peer components. The client only performs a traditional request-reply communication with a peer of its organization, *extending* the process with an additional initial request and final reply step. However, the underlying consensus mechanism remains unchanged; the functionality is mostly just a code reorganization. Accordingly, this paper assumes gatewayless operation.

In comparison to other platforms, notably Ethereum, this approach to blockchain consensus changes the order of transaction *ordering* and *execution*. As such, the above process is commonly called *Execute-Order-Validate* (abbreviated as EOV or XOV) consensus. While some other blockchains also follow the EOV pattern and its variants, we focus on Hyperledger Fabric in this paper due to its maturity and practical significance.

It is to be noted that the smart contract execution model of Fabric is not virtual machine but interface-based (in contrast to, e.g., Ethereum). Chaincode bundled with its execution environment either resides in a Docker container on the peer or its execution is fully detached as an external service. Chaincode communicates with the peer during

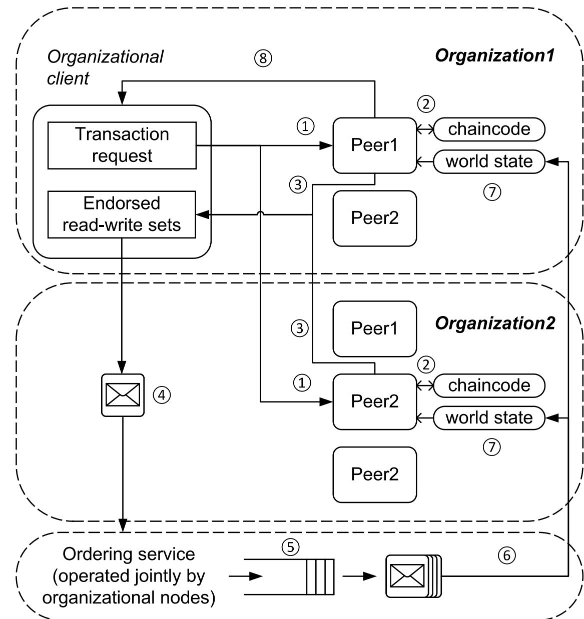

FIGURE 1. Transaction processing in a Hyperledger Fabric network.

transaction proposal execution through a key-value get/put RPC API.

One of the many ramifications of this model is that the chaincode development language, runtime, and execution environment can be almost arbitrary as long as the environment and executable can be encapsulated in a container, implement the RPC API, and adhere to a lifecycle model. Fabric natively supports JavaScript/TypeScript, Java, and Golang chaincodes.

B. MVCC CONFLICTS

In EOV transaction processing, read-write conflicts can arise between transactions due to their parallelized pre-execution. Multi-version concurrency control in Fabric handles these conflicts by invalidating some transactions during block validation on the peers; it processes the transaction read-write sets sequentially and discards all unsafe operations.

The core logic of conflict handling is simple: if, in a block, the pre-execution of a transaction relied on a variable – as expressed by the versioned read set – which has been updated in the world state since then it has to be discarded. The update may have happened during committing a transaction from the same block or could have happened even during committing an *earlier block*. Due to the nature and significant configurability of the EOV mechanism, end-to-end transaction latency can be significant (on the order of seconds), and it can be possible to simulate a transaction with a world state which will be updated shortly by a block already being prepared for broadcast to the peers. This way, we tend to distinguish *intra-block* and *inter-block* MVCC conflicts between transactions.

Reference [7] formalizes Hyperledger Fabric transaction failure modes and describes them in detail; in the terminology

of [7], this paper is primarily concerned with *MVCC read conflicts*. We introduce the new term **conflict horizon** to denote the age (as measured from initial request issuance) interval of transactions which a distinguished transaction may conflict with during its own (later) validation phase. The underlying intuition is that when a transaction is newly started, transactions over a certain age can be assumed to have already gone through validation and commitment (or invalidation) by the time the distinguished transaction begins to undergo endorsement. The conflict horizon does evolve during the lifetime of a transaction – during ordering, "too young" other transactions will not be able to interfere anymore (although the relationship is certainly asymmetric).

It is challenging to empirically characterize the risk associated with MVCC conflicts in the practice. Due to the consortial nature of Fabric networks, production chaincode, and workloads are very hard to find publicly (in stark contrast to Ethereum). That said, the experiments of [7], the publications we cite later as well as the professional experience of the authors indicate that it can be deceptively easy to create Hyperledger Fabric-based solutions where the goodput is markedly and unexpectedly lower than the throughput due to MVCC conflicts; especially when the ledger data model is not trivial (as, e.g., in [8]).

We will argue that for critical applications, application requirements can necessitate the systematic analysis and designed control of MVCC conflicts, irrespective of their unknowable incidence statistics across production Fabric deployments.

C. MOTIVATING EXAMPLE: WAYS TO HANDLE CONFLICTS

The state-of-the-art in MVCC conflict control (which, as a concept, we will define shortly) primarily focuses on the clients emitting transactions in an unfortunate manner and the transactions being ordered into blocks in a conflicting way. Only limited research (only the recent publication [9] seems to formulate an initial systematic approach) addresses the appropriateness of the way we map application concepts to the ledger key-value store to support a given business logic and workload profile setting. As some of our contributions investigate this latter aspect, we demonstrate this duality in a simple example, as depicted on Figure 2.

In our example, we track the attribute values of entities on the ledger. We opt for a straightforward key-value representation: an entity identifier serves as the key, and all attributes are stored in a value corresponding to that key on-ledger, e.g., as a JSON document. This approach echoes the standard encapsulation practice of object-oriented programming and seems to be standard chaincode development practice. On the figure, at time t_1, entity-key E has version v_1, storing the current value of attributes A_1 and A_2 on-ledger. A simple example we will use again later would be a person and their money accounts.

Transaction TX_1, accounting for a money transfer, aims to increase the balance of the first account. To that end, during endorsement, at t_1 the key is read on the endorsing nodes, the attributes parsed, and an endorsed read-write set created with $R = \{E^{v_1}\}$ and $W = \{(E, \{A'_1, A_2\})\}$, that is, the single key E is read and simulated to be written. The endorsements are bundled, sent for ordering, and then get back to the peers as a part of block validation; at t_3, the transaction is validated, and peer world states are updated to $(E^{v_2}, \{A'_1, A_2\})$.

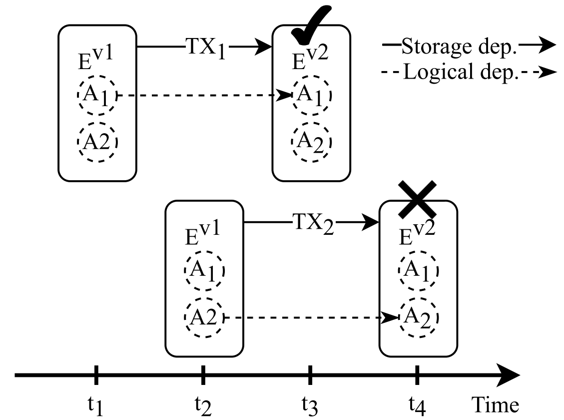

FIGURE 2. Temporal view of logically independent entity-attribute MVCC conflicts.

At t_2, between t_1 and t_3, a transaction TX_2 is endorsed to change the balance of the *other* account, but still using $R = \{E^{v_1}\}$, thus trying to update to $W = \{(E, \{A_1, A'_2\})\}$ instead of $\{(E, \{A'_1, A'_2\})\}$. At t_4, during block validation, MVCC conflict control recognizes that the write set was computed based on E^{v_1} while the committed version is already E^{v_2}; thus, TX_2 is invalidated and discarded.

While the literature widely recognizes the more mechanical categorizations of conflicts (as. e.g., read-write/write-write/write-read [10]) and the various transaction dependency graphs they give rise to, it has been mostly overlooked that there is a qualitative difference between the logical dependencies of transactions, rooted in transaction semantics, and storage-level dependencies, which flow from the way we choose to use the ledger key-value store for a given application.

Approaching the transaction conflict on Figure 2 at the storage level, we can try to eliminate the conflict through *temporal decoupling*: by manipulating the timing (including the global order) of transactions. For example, we can try to achieve $t_2 > t_3$ by decreasing the latency of transactions. We can also try to mitigate the conflict instead of resolving it: e.g., achieving $t_3 > t_4$ by reordering TX_1 and TX_2 in the ordering service if the commit of TX_2 would cause fewer overall conflicts than TX_1. We can also opt to abort one of the transactions early to avoid wasting network resources at least partially. In the next section, we will systematically review the relevant techniques.

At the same time, recognizing that we can minimize the read and write set overlaps between transactions while retaining business logic semantics, we can perform *data dependency refinement* by storing the individual accounts

under independent keys. This *prevents* the conflict depicted on Figure 2. However, semantics-based, conflict-minimizing key usage has received limited attention in the literature.

III. MEANS OF CONFLICT-CONTROLLED OPERATION

As practical experience in applying consortial blockchain to cross-organizational collaboration problems mounts, it is becoming increasingly viable to treat the design of systems with distributed ledger components through the lens of requirement-based system engineering.

A systematic design approach is warranted as consortial blockchains are at least moderately (business-)critical in at least integrity almost by definition; otherwise, a distributed ledger would not be necessary. As a logical corollary substantiated by practical experience, the need for a varying set of performance, dependability, and security guarantees can be expected to be present, too, simply because integrity is usually not the only concern in most critical systems.

Systematic, requirement-based design requires a corpus of at least semi-codified design knowledge. This encompasses functional and extra-functional requirement modeling; development, operational, and process models; hazards, risks, and the ways to mitigate them; and the testing, verification, and validation approaches that can demonstrate compliance to the requirements to the necessary extent.

Most of these foundational components of system engineering for blockchains are still missing or immature. As a contribution to the state of the art, in this paper, we propose a structured model for the *means of conflict-controlled operation in Execute-Order-Validate blockchains*, and specifically, Hyperledger Fabric.

By **conflict-controlled operation** we mean *the ability of a Hyperledger Fabric network to operate within some expected bounds on MVCC conflicts in a trustworthy manner*, with the help of appropriate system design and runtime mechanisms.

It is important to emphasize that we do not aim to give an overview of the means to guarantee performance, performability, timeliness, or Age of Information (AoI) [11] targets. Controlling for MVCC conflicts is a true sub-problem of design against requirements on these extra-functional properties.

- For *performance*, conflicts reduce goodput but do not influence the theoretical maximum of throughput.
- For *timeliness and AoI*, conflicts introduce problematic transaction retries but do not influence whether the block time is tuned correctly or whether a malicious ordering service can perform application-level attacks.
- MVCC conflicts can even be a consideration for *availability* at the application protocol level; if two kinds of transactions are consistently consecutive and conflicting, the one coming later will never be validated.

We propose a taxonomy of means for conflict control based on a proven lifecycle model that has decades of application in risk-based, requirement-driven design processes. We note in advance that *how* this taxonomy is best applied during design is a further step that needs investigation. However, dependability, the domain we borrow from, provides ample prior art.

A. A TAXONOMY BASED ON DEPENDABLE COMPUTING

For Hyperledger Fabric performance optimization, [9] proposes a multi-level optimization model, distinguishing the *user level*, the *data level*, and the *system level* and describes the BlockOptR tool for event log and process mining based optimization. While the user-data-system trichotomy is a good framework for the performance optimization of an already established system, we propose a taxonomy better suited to *designing* for conflict control, based on the widely accepted classic concepts of the dependability of computer and communication systems [12].

One of the common definitions of dependability is that it is *"the ability to avoid service failures that are more frequent and more severe than is acceptable"*. To further quote [12], a *service failure* is *deviation from correct service*; *errors* are (internal) *system states which may lead to subsequent service failure*; and *faults* are the *adjudged or hypothesized causes of errors*. In composed systems, the service failure of a system component is treated as an external fault from the point of view of the components relying on the failing one, giving rise to the notion of internal *error propagation* in a system.

We make the connection between dependability and MVCC conflicts by observing that when an EOV system has to invalidate a transaction at the last step of its life-cycle, then it is natural to treat the invalidation as a *transaction service failure*; the requesting client has to retry the transaction (or decide on some alternative action). The error state leading to the failure is two or more transactions being present in the system, carrying the possibility of one or more of them getting invalidated during block validation, and faults are such requests being initiated by clients that can cause such error states.

Note that the faults-errors-failures framework is apt from the determinism point of view, too; two transactions – e.g., one reading a key and another writing the same key – which are undergoing endorsement can be only *potentially* in conflict until at least one of them is put into a block. Ordering the transaction that writes the key *after* the transaction that reads the key does not lead to a conflict, while the opposite order does. Accordingly, the error state does not necessarily lead to a failure.

Interpreting conflicts as failures and potential conflicts as errors enables a systematic application of the existing consensus taxonomy of the means to attain dependability in our setting. In the following, we will make the distinction between potential and actual conflicts only when necessary for clarity. The fundamental means to attain dependability – fault prevention, fault tolerance, fault removal, and fault forecasting [12] – can be translated to MVCC conflicts the following way.

- *Fault prevention* translates to **conflict prevention**: preventing the appearance of potentially conflicting

transactions in the system to the required extent. Classically, fault prevention has been focusing on proper engineering practices and component selection; in our context, *workload schedule planning*, *load shaping*, and the possibly associated *admission control* can also be preventive measures.
- *Fault tolerance* translates to **conflict tolerance**: avoiding service failures – invalidated transactions – in the presence of potentially or actually conflicting transaction requests. Fault tolerance presupposes *error detection*; detected error states are processed through *recovery*, where error handling eliminates errors from the system state and optional fault handling prevents faults from being activated again. *Conflict detection* and *recovery from conflicts* are directly meaningful in our setting.
- *Fault removal* translates to **conflict potential removal**: reducing the number and severity of potential conflicts in the issued transactions. Fault removal can happen either during development or at runtime as corrective or preventive maintenance; the same can be said for conflicts. However, empirical evidence on the overall role of maintenance in cross-organizational blockchains is still lacking.
- *Fault forecasting* translates to **conflict forecasting**: estimating the present number, the future incidence, and the likely conflict consequences of transactions issued with a conflict potential.

Although faults, errors, and failures usually have a direct and nontrivial impact on performance, the classic means to attain dependability are not directly concerned with performance, as performance falls outside the set of attributes comprising the codified notion of dependability. At the same time, the task completion problems arising from depleted capacity reserves in a system – i.e., overload situations caused by *overload* or *capacity planning* faults – are *failures* in the strict sense and usually cannot be *recovered* from. Rather, overloads require a more permissive approach: the error state is not recovered from, but *mitigated* by "*masking errors and compensating for their effects*" [13, p. 179].

Reference [13] presents software and system pattern languages for error recovery as well as error mitigation. Generally, while recovery techniques roll back or roll forward an erroneous state to a non-erroneous one or fully compensate for it through redundancies, mitigation techniques focus on a graceful (partially compensatory) response to transient error states. Common patterns include *shedding* – refusing or aborting – *workload*; equitably allocating resources between workloads; deferring work; and, when possible and effective, expanding processing resources.

Mitigation is applicable to errors different from overload, too. MVCC conflicts are a prime example: if it can be detected early enough – e.g., during endorsement – that the invalidation of a transaction is unavoidable, then it makes more sense to abort it as early as possible to avoid unnecessary resource usage and to signal failure to the requesting client as early as possible. Thus, we also include the category of conflict mitigations.

- *Error mitigation* [13] translates to **conflict mitigation**: addressing conflicts and potential conflicts at runtime without fully recovering the expected normal operation; that is, executing and finalizing all valid transaction requests.

Note that the terminology in the literature addressing MVCC conflicts is not consistent; what we mean by *recovery* is sometimes described as *mitigation*. This is a common phenomenon with the concepts of dependability in general; related fields regularly use them in divergent ways or are much laxer with respect to terminological precision. We hope that our adaptation of the core concepts of dependability can facilitate more systematic assurance process thinking than the state of the art.

B. EXISTING APPROACHES

In this section, we provide a categorization of the known techniques for conflict-controlled operation in EOV blockchains, emphasizing Hyperledger Fabric. To identify the relevant approaches, we applied the following literature research strategy.

1) *Establishing a core set*: using Google Scholar with a wide search for the terms "MVCC conflict" and "Hyperledger Fabric", in 2022, [5] established a core set of existing MVVC conflict control techniques.[3] One-step forward and backward reference-chainings (limited snowballing) were also performed. The search was performed for "performance optimization" and "Hyperledger Fabric", too, but only the very limited set of results where the connection between performance and MVCC conflicts is made explicit was retained. All searches were cut off at 2018, as older papers could be realistically based only on the earlier, starkly different (non-MVCC) Fabric architecture.
2) *Follow-up reference chaining*: for this paper, using Google Scholar again, we performed forward snowballing on the reference set of [5] for 2022 and 2023.
3) *Cross-referencing with systematic analysis*: we became aware of [9] through Google Scholar recommendations on its publication. We cross-referenced it with our references (not finding significant relevant deficiencies on our side).

These three steps covered the conflict prevention, tolerance, and mitigation categories and provided an entry for conflict potential removal.

4) *Exploratory literature research for new categories*: In comparison to [5], *conflict potential removal* and *conflict forecasting* are entirely new taxa derived from a dependability-motivated approach, and not just a logical restructuring. For these categories, we reviewed Google Scholar results (from 2018 on) for the combinations of the search term "Hyperledger Fabric" with

[3] In this work, we propose a fully reworked taxonomy compared to [5].

"MVCC conflict analysis," "write conflict analysis," "read conflict analysis," and "conflict forecasting." The exploration was based on titles and abstracts and cut off after the first ten pages of hits (where applicable); due to the lack of relevant results, meaningful snowballing could not be performed.

As a general rule, we discarded all papers not discussing MVCC conflict control or providing relevant insights and those not published either in a peer-reviewed journal or in the proceedings of a peer-reviewed conference. In general, we excluded approaches not targeting Fabric, or only conceptually, but not technically being based on Hyperledger Fabric, such as [14].

Table 1 presents the results of our literature survey efforts categorized according to our taxonomy and laid out in a tabular form. Note that the repetitions in the table are intentional; many approaches are naturally composite, especially those that provide conflict tolerance for *some* conflicting requests and perform early aborts for the remainder.

C. CONFLICT PREVENTION

Conflict prevention can be performed from the point of view of numerous aspects, which we synthesized based on [5] and [9]. Although this is where the potential for some genuinely robust solutions lies from a system engineering point of view, important avenues of conflict prevention are far less developed than the techniques addressing potential and actual conflicts.

1) ARCHITECTURE AND NETWORK

At the platform software layer, significant changes to the Fabric architecture either facilitate the use of conflict-free datatypes [18] or lower transaction processing delay [16], [17]. Controlling conflicts through network configuration and sizing is also an indirect effect through lower end-to-end delays [7], [9].

The latter mechanism has been thoroughly investigated in [7] by a series of "tuning" experiments. From our point of view, the main insight of [7] is that irrespective of the workload, the configuration parameters that yielded the lowest end-to-end latency resulted in the fewest conflicts. Speedup can act as a temporal decoupling mechanism for conflicts: lower latencies mean earlier final transaction commits (after validation), leading to clean data reads by previously parallel but now subsequent transactions. In [7], the following tuning aspects were investigated.

- *Block time*: the time between creating two blocks. Although examined through changing the block size by [7], block time is a better-known metric in the blockchain domain. Lower duration thresholds lead to earlier block cut times, decreasing the waiting times and – consequently – the end-to-end latencies of queued transactions.
- *Endorsement policy*: the number of organization endorsements required for eventual transaction approval. Demanding fewer endorsements (i.e., parallel transaction executions) can shorten the endorsement gathering time (e.g., in geographically widespread systems) and decrease the overall load on the peers (also performing validations), both ultimately lowering the end-to-end latency of transactions. Note, however, that the endorsement policy is not just a performance-tuning parameter; it directly expresses the approach taken to trust decentralization.
- *Network complexity*: the number of organizations and peer nodes in the network. Network complexity largely shapes endorsement policies and node-to-node communication overheads, contributing to end-to-end transaction latencies.
- *State DB type*: either CouchDB or GoLevelDB can be used as the world state database for the network. The more performant GoLevelDB option can lower end-to-end transaction latencies due to faster endorsement and validation times.

In Table 1, we chose not to enumerate all known performance tuning parameters and only demonstrate their diversity.

2) DATA MODELING AND CHAINCODE ARCHITECTURE

There is only very scarce work on Hyperledger Fabric smart contract design with an explicit intent to prevent MVCC conflicts and some of it is only available as non-peer-reviewed, online material.

In the keystone Hyperledger Fabric paper [3], the performance case study uses a UTXO-based "FabToken" asset type. As UTXO-like constructions explicitly "track the money" instead of account balances, they enable multiple send and receive transactions for the same party in each others' conflict horizon - without MVCC conflicts. The downsides are the same as with all UTXO-based approaches (including the ledger not maintaining balances explicitly).

In [15], a *readless write* approach is followed; new ledger records are created in a log entry-like manner, and for transactions where up-to-date status information is necessary, logical rollups (a combination of range queries and aggregation) are used. This technique can dramatically reduce the number of conflicts, but its usage is limited to use cases with infrequent reads. In [9], this technique is formulated as a *delta writes* recommendation for writes where differential updates are meaningful.

Reference [9] also suggests "smart contract partitioning" (domain-level partitioning of the assets and corresponding functionalities). *Data model alteration* is also specified as a category, but only a few examples are given (the logic of which is covered by Section IV).

3) BUSINESS PROCESS

The authors in [9] point out that for Hyperledger Fabric use cases with cooperation process semantics, such modifications at the process level as model pruning and activity reordering can lead to the prevention of conflicts.

TABLE 1. Means of conflict-controlled operation in Execute-Order-Validate (EOV) blockchains: taxonomy and known solutions. *: indirectly related.

Means	System aspect/EOV phase	Proposed solutions	Base Fabric compatibility
Conflict prevention	Client workload	Rate control [9]	Fully transparent
	Business process	User activity reordering [9] Process model pruning [9]	
	Data model and storage mapping	Bitcoin-style UTXO [3] Readless writes and aggregation [15] Delta writes [9] "Data model alterations" [9]	
	Chaincode software architecture	Smart contract partitioning [9]	
	Network config. and sizing: higher processing rate can lower conflict probability	Block sizing [9][7] Endorser restructuring [9][7] Client resource boost [9][7] Other latency optimizations [7]	
	Architecture	FastFabric [16]: speedup lowers confl. prob. StreamChain [17]: speedup lowers confl. prob. FabricCRDT [18]: conflict-free datatypes	Significantly different from base Fabric
Conflict tolerance	Client endorsement request	LMLS [19]: delta trans. based on client lock cache	Cache locality unclear
	Endorsement		
	Client endorsement dispatch	[20]: client-local cache and reexecution-mgmt	Partial solution
	Ordering: transaction reordering techniques	Fabric++ [10]: dependency-based reordering FabricUp [21]: algorithmically improves Fabric++ FabricSharp [22]: improves Fabric++ abort rate FabricETP [23]: improves time complexity CATP-Fabric [24]: also prioritization and merging	Major modifications As Fabric++ Modified ordering Modified ordering Modified ordering
	Validation	XOX Fabric [25]: reexecution during validation	XOV → XOX
Conflict mitigation	Client endorsement request		
	Endorsement	EMVCC [26]: early abort after endorsement Fabric++ [10]: early abort during endorsement	Endorser-local caching Major modifications
	Client endorsement dispatch		
	Ordering: early abort techniques	Fabric++ [10] FabricUp [21] FabricSharp [22] FabricETP [23] CATP-Fabric [24]	Major modifications As Fabric++ Modified ordering Modified ordering Modified ordering
	Validation	Default MVCC conflict handling	
Conflict potential removal	Both during development and use	BlockOptR [9]: recommendations from log mining AdaChain* [27]: arch. selection with reinf. learning Athena* [28]: auto-tuning HLF with reinf. learning FSC* [29]: static analysis and dynamic execution	Fully applicable XOV is just one option Conflict control implicit Currently Ethereum
Conflict forecasting			

4) CLIENT WORKLOAD

Transaction rate control can also be a tool for conflict prevention, as pointed out in [9], however, without discussing its specific challenges in the distributed ledger context, specifically for Hyperledger Fabric.

First, rate control either requires the cooperation of the clients of the participating organizations, or the rate control measures have to be implemented as early abort mechanisms in the endorsing smart contracts. A fuzzy logic-based variant of the latter has been proposed in [30] and [31], although without explicitly considering conflicts. Without smart contract-based enforcement, an organization can get an unfair advantage over others by disregarding its own rate limits. On the other hand, this variant of "dishonest behavior" is easily detectable in Hyperledger Fabric.

Second, production-grade rate control logic design can be a rather involved challenge, as it may have to implement an *equitable resource allocation* [13] across organizations and transaction types of the block space as a limited resource. Currently, the algorithmic support for such requirement-driven admission control design seems missing for Hyperledger Fabric.

D. CONFLICT TOLERANCE

Conflict tolerance mechanisms eliminate or reduce the conflict effects of potentially conflicting transactions without

aborting them. We subdivide the mechanisms based on the Fabric consensus stage where they act. As already mentioned, many mechanisms appear in the conflict mitigation category, too, as in most cases, some transactions can remain in conflict; for those, it is still more appropriate to abort them early than waste more resources by finishing the entire Fabric transaction lifecycle.

1) CLIENT ENDORSEMENT REQUEST

Clients can choose not to start transactions based on the outstanding ones or issue *different* transactions. Reference [19] creates delta-writes based on its local cache. However, it is unclear whether the cache is shared between at least the clients of the same organization, and the same unfair behavior considerations as with rate control still seem to apply.

2) ENDORSEMENT

We did not find any approach that would introduce conflict *tolerance* at the endorsement stage. However, it is logically possible in a minimally invasive way from the point of view of Fabric consensus and platform software if there is *redundancy in time* with respect to executing the transaction requests.

Namely, if an endorsing peer can recognize that it has already endorsed a transaction with which an incoming endorsement request would be in conflict, it can delay endorsement until it receives and validates the already endorsed transaction's block. At that time, an endorsement relying on the updated ledger values can be issued.

It can even be possible to waitlist requests without executing them, preserving resources if request-pair conflict probabilities can be learned or determined during a model-based system design process. Such a solution would require a cache of endorsed but not validated transactions shared across the endorsing peers of each organization, but given the usual limited horizontal scaling inside individual organizations (a dozen peers is usually considered large-scale), this is easily achievable at sufficient performance with a distributed, in-memory, non-Byzantine fault-tolerant database.

In this paper, we only identify this gap – uncovered by our systematic approach – and outline an apparent solution. The report this paper is based on [5] provides a more detailed blueprint, but prototyping and assessing the solution is an open technical problem, and the learning aspect is an open research problem.

3) CLIENT ENDORSEMENT DISPATCH

Reference [20] presents an approach where clients cache their endorsements and decide to have some transactions re-endorsed instead of submission for ordering. Questions of unfair behavior seem to apply, and it is not obvious how effective the approach can be at the system level.

4) ORDERING SERVICE

Most approaches focus on modifying the order of endorsed transaction requests in the ordering service [10], [21], [22], [23], [24] to create a conflict-free transaction order (and to early abort the transactions for which this cannot be done). This is a well-researched area, and intervention in the ordering service has the benefit that it can be easily realized and maintained as an architecturally localized customization of the platform consensus mechanism. Our comment (and not critique) is that as these mechanisms act on the computed read-write sets, they should be employed with certainty gained at design time that they will be effective to at least some degree. We did not find any guidance on the pre-operation assessment of the cited techniques.

5) VALIDATION

XOX-Fabric [25] presents a unique conflict tolerance approach: instead of ''throwing them away'', transactions are *reexecuted* during validation, giving rise to a new, ''Execute-Order-Execute'' computational model. This is an elegant solution, although it modifies Fabric's consensus semantics in important ways (e.g., for the re-execution, in contrast to normal endorsement, a client cannot choose not to proceed with a transaction based on the execution result). Also, the post-order execution needs a dedicated ''patch-up code'' in the smart contract, and the keys in the new read-write set must be a subset of the original (invalidated) one. Smart contract logic can still prescribe transaction rejection (e.g., trying to spend from an account emptied by an earlier transaction, which led to the invalidation in the first place).

E. CONFLICT MITIGATION

Conflict mitigation approaches abort transaction processing early, potentially protecting resources and facilitating earlier retries. The same categorization to consensus phases can be applied as with conflict tolerance, and due to the noted overlaps, we outline only the differences.

EMVCC [26] performs early aborts after endorsement, using endorser-local caching, and one mechanism in Fabric ++ [10] is early abort during endorsement. In our view, mitigation at the client endorsement request phase does not make sense logically; ordering incorporates the same approaches as conflict tolerance, and mitigation during validation is effectively the base Fabric MVCC conflict-handling behavior.

F. CONFLICT POTENTIAL REMOVAL

We defined conflict potential removal as reducing the number and severity of potential conflicts in the issued transactions, denoting an activity performed during development or system maintenance.

The relevant literature for Hyperledger Fabric is scarce; BlockOptR [9] seems to be the only directly applicable approach.

Indirectly, emerging architecture selection and Hyperledger Fabric auto-tuning tools, such as AdaChain [27], and Athena [28] may be applicable, at the very least through reacting to conflicts by trying to compress the transaction conflict horizons. However, all of these are empirical methods; their

use at development time presupposes implementation and the availability of expected load profiles to the specificity where the load can be generated with tools like Hyperledger Caliper.

As an outlook, we note that parallelization efforts for the Ethereum Virtual Machine (EVM) (see, e.g., [29]) have been introducing fine-grained ledger variable read-write analysis through a mix of static smart contract analysis and dynamic execution. These approaches may prove adaptable to remove potential MVCC conflicts under transaction request patterns that are either expected or enforced.

G. CONFLICT FORECASTING

We did not find any relevant specific literature for conflict forecasting; we hypothesize that an explicit connection between the workload at the semantic level and the reads and writes of workload items (Section IV) can enable the prediction of the evolution of MVCC conflicts, including emerging "conflict storms".

H. GAPS IDENTIFIED

We believe that our taxonomization of the existing approaches for conflict-controlled operation in Fabric points toward the following gaps in the state of the art.

GAP 1. The mapping between the domain data model and its operations to chaincode-computed key-value changes lies at the heart of all conflict prevention approaches, which do not require any change to Fabric (apart from reducing end-to-end delay). However, this mapping has not been proposed yet to be utilized systematically as a part of designing for conflict prevention.

GAP 2. The possibility of conflict tolerance during endorsement is not utilized.

GAP3. In dependable computing, the means to attain dependability act in concert. Governed by the level of system criticality and the extra-functional requirements, fault prevention and removal are complemented by systematically choosing and evaluating fault tolerance and mitigation mechanisms during design. Fault removal and forecasting also extend to operation time. There has not been a similar model of conflict-controlled operation for MVCC conflicts, and there is no guidance on the co-application of the existing techniques during the system lifecycle.

We make conceptual and technical contributions to GAP 1 in the upcoming sections. Earlier, we briefly outlined a technique that can address GAP 2. We made an initial contribution to GAP 3 with our taxonomization, but at the same time, recognize that much work is remaining.

IV. TOWARDS MDE-BASED CONFLICT PREVENTION

Reference [9] treats MVCC conflicts as a part of performance and approaches them from the point of view of measurement-based optimization. We propose that it is also possible to consider MVCC conflicts from the initial system design phase, and this constitutes an *outstanding gap in the conflict prevention category* which *can be addressed with Model-Driven Engineering (MDE) techniques*.

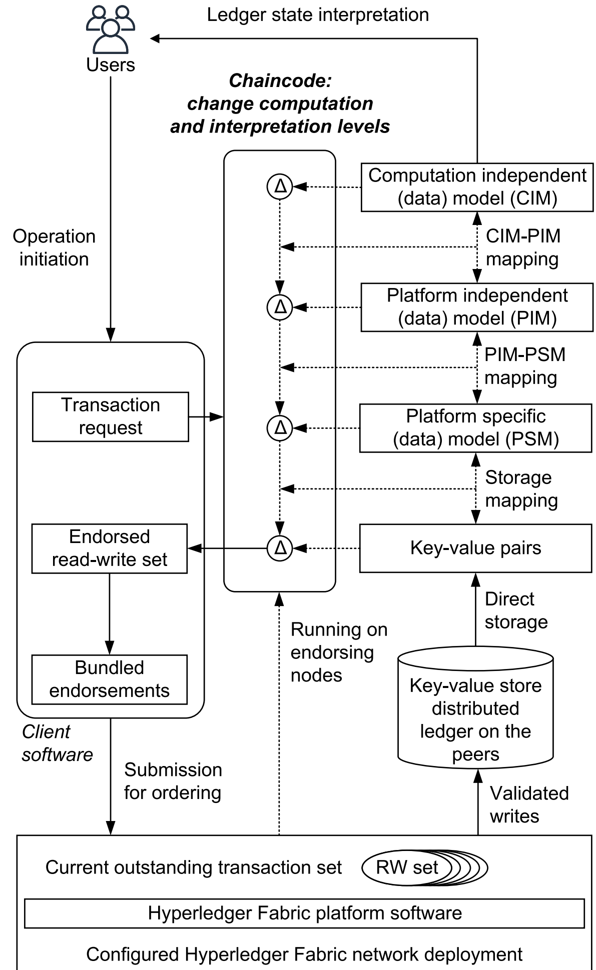

FIGURE 3. A ledger change interpretation-oriented view of Fabric transactions: chaincode-computed ledger changes (Greek delta) correspond to a series of interpretations at different abstraction levels.

A. THE SEMANTIC GAP

To guide our discussion, Figure 3 presents a ledger change interpretation-oriented view of Hyperledger Fabric transaction execution. In the interest of legibility, the figure abstracts away the endorsement by multiple nodes aspect of the process.

In Fabric, there's a form of a *semantic gap* between the business-level interpretation of ledger data and operations and the basic versioned key-value ledger data model it supports. There aren't even typing and data shape constraint enforcement facilities – such as database schemata in relational databases – unless chaincode implements them. Consequently, mapping the *intent* of ledger changes in terms of the business interpretation of the ledger to key-value changes is entirely left to the chaincode implementation.

From the system engineering point of view, in the majority of consortial blockchain use cases, the ledger carries and manages either the state of collaboration processes or "assets" in the most generic sense. A review of Fabric use cases is beyond the scope of this paper; as supportive

evidence, we refer to the Hyperledger use case tracker[4] and as an analogical argument, we point out that the same can be said for the decentralized applications implemented on public blockchain networks [32].

Without the blockchain context, the notions of the collaboration process and asset state induce *ontological* concepts and relationships over them; and, as in a wide variety of use cases, a distributed ledger acts *functionally* as a database with special *extrafunctional properties*, we can assume that from the decision making and business process point of view, users interpret ledger content in largely blockchain-independent terms. The semantic gap manifests in the high-level *meaning* of ledger content and its changes having to be "serialized" to a key-value pair-based storage approach.

In system and software engineering, when such relationships are nontrivial, it is usually expedient to express them as a series of *model refinements*. The well-known base mechanics of Bitcoin can give us a simple example here: in the abstract sense, users usually interpret ledger state as "how much Bitcoin they have"; what at the platform-specific level translates to the *concept* of unspent transaction outputs (UTXOs) existing on the ledger which are spendable with their private keys; which, in turn, have a specific transaction output representation in mined Bitcoin transaction blocks.

B. THE SEMANTIC GAP IN MDE TERMS

Due to the increasing role of Model-Driven Engineering (MDE) in smart contract development, which we expand on later, we use the model hierarchy of the *Model Driven Architecture* (MDA) from the Object Management Group (OMG) [33] to characterize the relationship between distributed ledger interpretation and implementation.

- A *Computation Independent Model* (CIM) does not show system-specific details; it is familiar to the practitioners of the application domain, and it is often called a *domain model* and *vocabulary*.
- A *Platform Independent Model* (PIM) is already a "system view" but is not dependent on the concepts of a target platform.
- A *Platform Specific Model* (PSM) incorporates the platform concepts and defines how a system uses a specific platform.

MDA, when applied for system design and development, advocates creating a series of model refinements from CIM through PIM(s) and PSM. The PSM is the direct basis of implementation. Modern MDE connects refinement levels with metamodel-based, automated model transformations; applies formal methods for verification and validation at various levels (creating the input of analysis tools with model transformations and marking back the analysis results to the model); and uses at least partial code and artifact generation in the PSM-to-implementation step. Note that MDE, in general, encompasses several approaches different from MDA [34], too, and the field is constantly evolving; however, for our purposes here, classic MDA with its codified model refinement approach is a good fit.

In terms of the CIM-PIM-PSM framework, the *intention* of a chaincode-implemented transaction is to compute the necessary changes (denoted with Δ on Figure 3) in the CIM; which correspond to changes in the PIM(s) and the PSM; which correspond to changes and accesses in key-value pairs, giving rise to the computed read-write set which is created as a chaincode execution result. The correspondence of the changes (and possible model "navigations" necessary to compute them) are governed by the mapping relations between the implementation and the models.

This is a *logical* framework connecting the tangible results of chaincode execution with their domain semantics using the MDA abstraction hierarchy. Apart from the most straightforward applications, this abstraction hierarchy always exists, even if it is fully implicit – i.e., when developers directly write chaincode, which serializes/deserializes ledger key-value pairs from/to chaincode-local program variables and objects in an ad-hoc way.

We posit that deriving the key-value storage model along an explicit, MDE-style model refinement hierarchy facilitates *conflict prevention* at design time. We present an outline of setting up the CIM-PIM-PSM-implementation refinement hierarchy for this purpose and the basic logic of refinement pattern application. On this basis, Section V describes a novel technical contribution covering the PIM-PSM-implementation fragment.

C. MODEL REFINEMENT BASED STORAGE DESIGN

For a consortial distributed ledger, where consensus participation and access to the ledger services are permissioned, the application data model, the transaction types, and the expected transaction load profiles are not unknown, as in public blockchains. The network is created for a specific business cooperation purpose, and users do not deploy new smart contracts at their will. Instead, smart contract (chaincode) deployments and updates are controlled change processes that the participating organizations perform cooperatively. Additionally, for any structured system development process, these aspects must be captured as requirements in the early stages; there is no reason for blockchain-based solutions to be exceptions.

1) CIM-PIM MAPPING

Given this premise, it is possible to map a domain model to an appropriate PIM based on the expected *logical* read/write accesses in the expected CIM model spaces and a *general* understanding of the nature of transaction processing.

Figure 4 outlines an example for persons and their (money) accounts. The critical transaction is transferring money across accounts; legal name and address changes are expected to be infrequent. Although we do not necessarily know the specific platform at this stage yet, at least the computational model is specific for a PIM; i.e., we do know that transactions

[4]Available at: https://www.hyperledger.org/learn/use-case-tracker, last accessed: 11.08.2023.

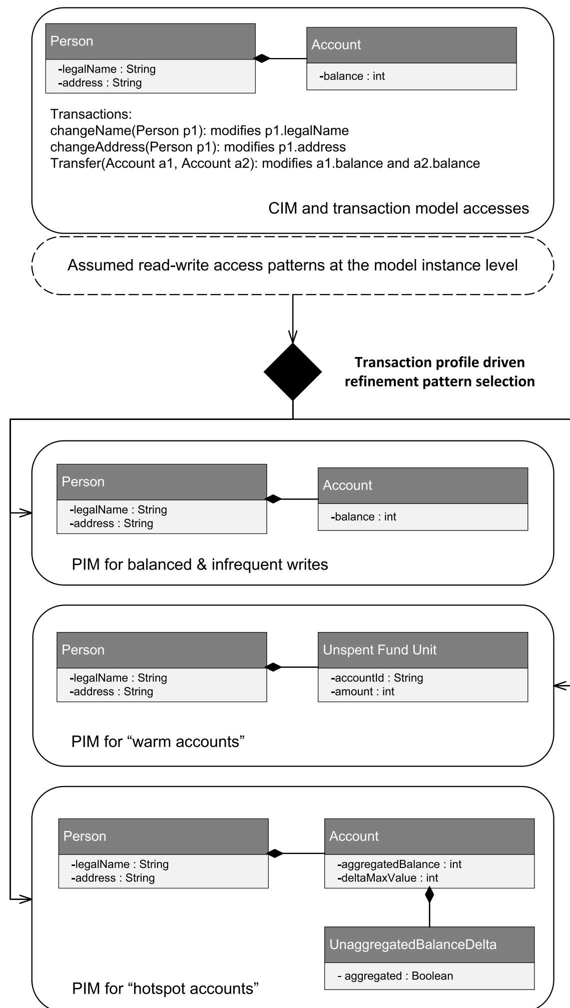

FIGURE 4. From CIM to PIM in ledger data modeling: an outline of three transaction profile driven strategies.

will be executed on a blockchain. Blockchains, by definition, use transaction batching; thus, we may want to create a PIM that minimizes transactions' reliance on unfinalized modifications.

If we know that we can expect a transaction on an account to appear mostly once on an appropriate event horizon, then we can reuse our simple CIM model. A good example would be a retail central bank digital currency if it is offered only to citizens and small businesses. These users can be expected to have at most one incoming and outgoing transaction in a few blocks' time if the block time is low enough (for Hyperledger Fabric, the default is 1 second).

Conversely, we may know that we have "warm accounts" in that a significant ratio of accounts regularly has multiple incoming and outgoing transactions on an appropriate time horizon. In this case, a UTXO-like solution will be more appropriate.

Lastly, if we know that we have to support at least a few "hotspot accounts" – in the extreme, all transactions targeting them for some periods – then we may opt to use a delta write + aggregation pattern. This certainly requires overspend protection for a money use case; the model indirectly controls overspends by limiting maximum transaction value.

2) PIM-PSM MAPPING AND IMPLEMENTATION

Given a PIM, for Hyperledger Fabric, the most straightforward PIM-PSM mapping specifies the way we *partition* the properties of entities across keys at the logical level. (A more complete PSM should include further platform concepts like channels and private data collections.) Figure 5 presents an outline of three basic mapping strategies. The PIM is the first option from Figure 4.

In the first case, the person's accounts (note the composition relation) are grouped into a single composite value. This simplifies chaincode development and ledger consistency management – chaincode can navigate to "everything belonging to a person" in a single call. However, this approach introduces the possibility of transaction conflicts, which can be undesirable and *could be avoidable*: e.g., two different accounts of the same person receiving transfers in the same conflict horizon.

The second option addresses this drawback with a *partial* partitioning. Note that we show in the second example a concept partially orthogonal to partitioning, too: storing an association in the person value enables chaincode to read the account key from the person and navigate to the account atomically.

The third example shows a fully partitioned model.

Last but not least, a PSM translates to a specific key-value storage approach. Even in this step, there is design variability; in the example, we use very simple key composition schemes and JSON values.

3) METHODOLOGICAL HYPOTHESES

In this paper, our methodological contribution to MDE-based conflict prevention is the above example-supported approach outline and the formulation of the following hypotheses.

- For Hyperledger Fabric, given the nature of Fabric-based systems and the fact that, for critical applications, their development can be expected to follow a proper design process, it is possible to reason about data access conflicts well before (full) implementation in a systematic way if MDE principles are also followed.
- Specifically, after a CIM-PIM mapping, assessing logical data access conflicts (model reads, navigations, and writes of transactions) can provide a pessimistic overabstraction of the actual MVCC conflicts in implementation.
- Given an (expected) transaction workload profile at the semantic level, a specific PSM can serve as the basis of preliminary MVCC conflict analysis.
- On the constructive side, MDE-based conflict analysis and arguments support the creation of a maximally conflict-preventing data storage implementation.

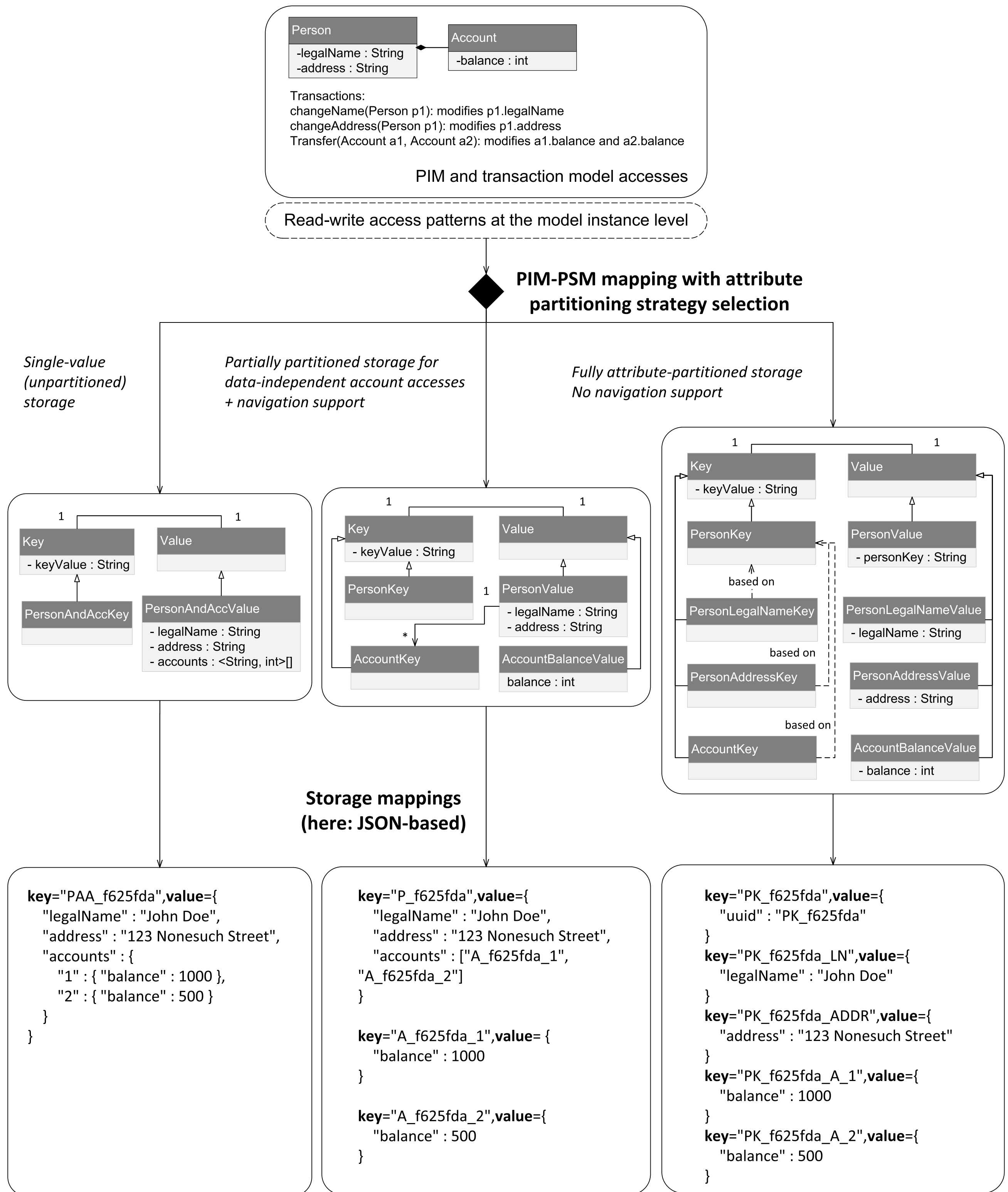

FIGURE 5. From PIM to PSM in ledger data modeling: an outline of three transaction profile driven strategies.

These hypotheses are certainly in need of validation. Implementing and evaluating an end-to-end, MDA-inspired conflict prevention approach still has numerous missing parts and is part of our ongoing research. However, to our knowledge, the approach outlined above is entirely novel, and Section V makes a tangible contribution to the fourth point.

D. TRANSACTION CONFLICTS AND MDE FOR FABRIC

Using MDE for blockchain is not a new concept, especially in its more simple manifestations when it is applied in the form of a single-stage code generation from a general-purpose modeling language or domain-specific language. That said, the potential to support design for conflict prevention and systematically designing conflict control for Hyperledger Fabric seems not to have been recognized yet.

To support this claim, we have performed a targeted literature review based on [35], a recent and comprehensive structured literature review on the design of blockchain-based applications using MDE and low-code/no-code platforms.

We filtered the cited conceptual methods [35, p. 11] as well as executable code generation methods [35, pp. 12-13] to the ones identified as applicable either to Hyperledger Fabric or multiple platforms. We performed forward citation chaining from this seed set (and based on the content of the papers, one-step backtracks among their references), taking into account peer-reviewed papers.

We present our negative result in three categories: grouping MDA-like approaches (where there is at least multi-aspect modeling or consistent use of hierarchical modeling) and approaches targeting the Business Process Model and Notation (BPMN) language. The third category is miscellaneous.

1) MDA-LIKE APPROACHES

iContractML 2.0 [36], an extension of iContractML [37], defines a domain-specific language that acts as a PIM reference metamodel and supports smart contract generation to a variety of platforms. The metamodel covers data (assets), transactions, and participants. Fabric is supported indirectly by code generation for Hyperledger Composer (now defunct) and DAML – both smart contract languages for which interpreter chaincodes exist.

Reference [38] synthesizes chaincode and other artifacts for hybrid on-chain/off-chain applications from an ensemble of models, including a domain, an action, and an interaction flow model. However, the "Ledger Object Generator", the critical component from our point of view, is not discussed.

Reference [39] presents automatic smart contract generation for Hyperledger Fabric from the DEMO (Design and Engineering Methodology for Organizations) language. The process involves defining a "fact model" (a relational domain model) and compiling an "Action Model" to Go chaincode. The relational domain model seems to be transformed to a Fabric storage scheme on a per-relation basis (all row field values serialized to a single JSON object).

To our knowledge, the recent MDAsmartCD approach [40] is the first attempt to fully apply the MDA life cycle. MDAsmartCD sets up a full CIM-PIM-PSM model chain and uses model-to-model transformations. However, neither the key-value mapping aspect of data modeling nor conflicts are first-class concepts. Reference [41] presents a precursor to MDAsmartCD.

MDE4BBIS [42] is similar but applies only a PIM-PSM approach. Reference [43] also focuses on the PIM-PSM step, starting from ArchiMate and targeting the (now defunct) Hyperledger Composer chaincode domain-specific language and middleware.

None of the above papers investigate either performance or MVCC conflicts.

2) BPMN

Supporting cross-organizational collaborations is one of blockchain technology's dominant non-crypto applications. As BPMN is the most widely used modeling language to capture business processes across organizations, it is natural that there is significant interest in creating blockchain support for collaboration based on BPMN models. The general state of the art is much broader than what we touch on; we restrict our discussion to approaches relevant to Hyperledger Fabric.

Mantichor [44] is a multi-chain architecture for business process choreographies with code generation from BPMN. Performance and conflicts on Fabric are not discussed.

Multi-Chain [45] also addresses multi-party choreographies on multiple blockchains (but not cross-chain), including Hyperledger Fabric. Initial and inconclusive performance figures are reported, but the approach seems to serialize the entire choreography state to a single key. This limits the frequency of actions on each choreography to one for each conflict horizon.

TABS [46] transforms BPMN models into discrete event Hierarchical State Machine smart contracts, enabling parts of collaborations to be offloaded for side-chain processing. Basic latency characteristics are investigated, but not throughput (and conflicts on execution on Hyperledger Fabric).

Reference [47] models inter-organizational collaborations as BPMN process models, transforms them to minimized organizational state charts, and generates code for a custom state chart execution engine chaincode through State Chart XML (SCXML) as an intermediary language. Performance and conflicts are not addressed.

3) OTHER APPROACHES

Reference [48] presents a Hyperledger Fabric-based solution for executing business rule sets under a mixed on-chain/off-chain model. Explicit domain modeling is part of the approach; world-state accesses are executed by the business object integrations of the rule execution engine deployed in the chaincode execution container. Reference [49] generates Go chaincode from domain-specific OWL ontologies and SWRL rules. Reference [50] generates chaincode from an ontology-based model of legal contracts, which also covers domain data modeling. References [48], [49], and [50] discuss neither conflicts nor performance.

Reference [51] introduces a domain-specific smart contract language that combines linear dynamic logic formulas to be enforced and business rules. Contracts are compiled to SCXML notation and executed in Hyperledger Fabric on a

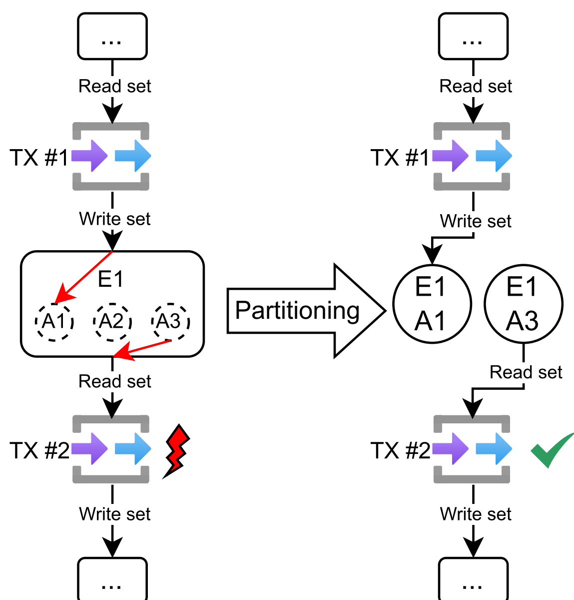

FIGURE 6. The core idea behind storage-level partitioning.

chaincode container deployed SCXML interpreter. Only very rudimentary performance measurements are reported.

4) SUMMARY

Emerging MDE and low-code/no-code approaches for chaincode creation for Hyperledger Fabric seem to be entirely unconcerned with explicitly supporting high-goodput operation and preventing MVCC conflicts at this point. For specific domains, and somewhat more generally, for business processes, this can be justified by expected orders of magnitude difference between transaction-on-entity and block frequencies. However, this assumption does not hold in general; even business processes may express high-frequency, automated activities (although we did not find this justification stated). Thus, we believe that MVCC conflicts and their avoidance, if the need arises, should be – at the very least – a recognized aspect of Fabric-targeting MDE.

V. CONFLICT PREVENTION WITH STORAGE-LEVEL ENTITY PARTITIONING

Having introduced entity attribute partitioning as a concept in the context of ledger data PIM-PSM mapping in the previous section (see Figure 5), we propose two specific "storage level" partitioning algorithms in this section. We also describe a chaincode SDK prototype that supports developing chaincode against PIM-level concepts and transparently implements serialization to the appropriate key-value level operations.

Expanding on Figure 2, on Figure 6, we show how storage partitioning can alleviate MVCC conflicts.

On the left side, transaction TX_1 modifies entity E_1 as part of its write set. Even though TX_1 only updates the value of attribute A_1, the entire entity is now "dirty" because every attribute of E_1 is stored under the same key-value entry in the ledger, as per standard object-oriented design practices. In the meantime, a second transaction – TX_2 – is also endorsed in parallel with TX_1, including E_1 in its read set. Even though TX_2 only needed attribute A_3 for its business logic calculations, it read all attributes of E_1 from the ledger. The entity-level storage mapping choice put the transactions into each other's conflict horizons, introducing a potential MVCC conflict that can lead to the invalidation of TX_2 if TX_1 is put earlier into a block by the ordering service.

Refining the unit of key-value storage from the entity level to the attribute level in the PSM (as depicted by the second and third columns in Figure 4) prevents potential MVCC conflicts by decreasing the conflict horizon of transactions. The right side of Figure 6 demonstrates the preventive effect: now transactions "dirty" only the actually used attributes of the entities, making the storage-level dependencies equivalent to higher level, logical dependencies.

If those still cause an unacceptable number of MVCC conflicts, then PIM-level approaches can also be applied transparently on top of the proposed PSM-level solution (in conjunction with other prevention and conflict tolerance/mitigation mechanisms).

A. PARTITIONING ALGORITHMS

As defined in [25] as the "*Hot Key Theorem*," if l is the average time between a transaction's execution and its state modification commitment, then the average effective throughput for all transactions operating on the same key k is at most $1/l$. By dividing the data stored under key k to $n \in \mathbb{N}$ parts and storing them under keys $\{k_1, \ldots, k_n\}$, the average effective throughput of all transactions that operated over key k becomes at most n/l, i.e., increases linearly with the number of parts. We present two partitioning algorithms, which differ essentially in choosing the partitions k_i.

- **Total partitioning** creates a separate partition k_i for every attribute A_i of every entity instance E_j, resulting in $n = |\{A_i\}|$ parts.
- **Attribute affinity-based partitioning** groups the attributes based on co-access patterns, resulting in potentially fewer, $n \leq |\{A_i\}|$ parts.

1) TOTAL PARTITIONING

Total partitioning maximally reduces cross-transaction storage dependencies by storing every attribute of every entity in a standalone partition. Thus, each attribute is stored under a unique key on the ledger, enabling MVCC with the highest possible granularity (on the PSM level). The algorithm for transforming the platform-independent data model to the platform-specific data model is simple, as it barely requires a data PIM. Another advantage of the approach is that its preventive effect is independent of the actual workload profile and its possible future changes (which might not be predictable at design time).

The drawback is that total partitioning potentially inflates read and write sets for complex transactions. Consequently:

TABLE 2. Example apriori data for transaction types (attributes accessed by transactions are marked with a 1, otherwise 0).

TX type	Attribute accesses			Load rate (TPS)
	A_1	A_2	A_3	
TX_1	1	0	1	30
TX_2	0	1	0	50

TABLE 3. Attribute affinity matrix created from table 2.

attributes	attribute1	attribute2	attribute3
attribute1	30	0	30
attribute2	0	50	0
attribute3	30	0	30

TABLE 4. The Attribute affinity matrix of table 3 after applying the Bond Energy Algorithm.

attributes	attribute2	attribute1	attribute3
attribute2	50	0	0
attribute1	0	30	30
attribute3	0	30	30

- the number of ledger accesses may increase, negatively impacting the endorsement latency;
- the size of the read and write sets may also increase, negatively impacting both the endorsement and block validation latencies.

However, a partition can be considered *unnecessary* if no cross-transactional storage dependency is ever resolved by its creation. For example, if attributes A_1 and A_3 (from Figure 6) are always updated together by every related transaction, their separation into different partitions is superfluous. Their "dirtiness" can be checked together by the MVCC mechanism, i.e., they can reside under the same ledger key.

2) ATTRIBUTE AFFINITY-BASED PARTITIONING

Eliminating such superfluous partitionings is analogous to the transaction data access-based vertical partitioning of traditional database schemas, a topic dating back to the 80's [52].

We will use the intentionally simple, example apriori transaction data in Table 2 to demonstrate our application of the algorithm presented in [52]. First, an attribute affinity matrix AA, a symmetric square matrix, is created from the input data, where each field contains a value quantifying the similarity of each attribute pair in terms of their access pattern. We use the summed load rate of functions accessing both attributes as their similarity value, as shown in Table 3.

In the next step, the *Bond Energy Algorithm* (BEA) [53] is used for diagonalizing AA, resulting in the diagonal block matrix AA_B of Table 4. Each block along the diagonal corresponds to a set of attributes that are similar to each other with respect to the employed similarity metric.

After the clustering step, the *SPLIT_NON_OVERLAP* binary partitioning algorithm from [52] is applied to the block matrix AA_B. The algorithm first splits the AA_B matrix at m different locations along the diagonal, where m is the size of the (square) matrix. Each split results in a different upper and lower block of AA_B, representing the binary partitioning of attributes. From all the binary partitions, only one is selected, such that the number of transactions that access only one of the two partitions is maximized while the number of transactions accessing both partitions is minimized. This is done by finding the binary partitioning with the maximal z value

$$z = c_u c_l - c_i^2 \tag{1}$$

where

- c_u is the number of transactions accessing only the upper partition of the AA_B matrix;
- c_l is the number of transactions accessing only the lower partition of the AA_B matrix;
- and c_i is the number of transactions accessing both partitions of the AA_B matrix.

Only partitions with a positive z value are accepted. N-ary partitioning of the attribute set is achieved by recursively applying the binary partitioning algorithm to the affinity matrices of the resulting attribute sets.

3) THE INTUITION BEHIND ATTRIBUTE AFFINITY-BASED PARTITIONING

The original vertical partitioning algorithm was designed for efficient data access in databases. Accordingly, the semantics of the applied similarity metric incorporated many physical traits of the system (I/O performance, caches, data sizes, etc.) and included all forms of data access, i.e., reading and updating attributes. In order to use the algorithm for MVCC conflict prevention, we had to redefine the semantics of attribute access.

The goal of attribute grouping, in our case, is not to speed up access to attributes that are read and written together. Rather, the goal is to group attributes that contribute to potential MVCC conflicts in "a similar way." Defined succinctly, MVCC conflicts are caused by keys that became dirty since their last read. A key becomes dirty when a transaction updates it through a write operation.

Correspondingly, if two attributes are always updated together, then storing them under separate keys (i.e., partitioning them) is unnecessary because their partitions will always get dirty together; they behave identically from the point of view of causing potential MVCC conflicts. Thus, our apriori transaction and attribute data contains only the update of attributes by transactions, i.e., a (TX_i, A_j) cell in Table 2 is 1 if TX_i sets A_j during its execution, and 0 otherwise. Note that read-only transactions (i.e., queries) are served by peers directly, without the consensus mechanism, and they will correspond to rows having zero values only in the attribute affinity matrix. Thus, they can be eliminated from the input data altogether.

4) POTENTIAL DRAWBACKS OF AFFINITY-BASED PARTITIONING

The attribute affinity-based partitioning approach aims to improve the potentially wasteful (in terms of latency and

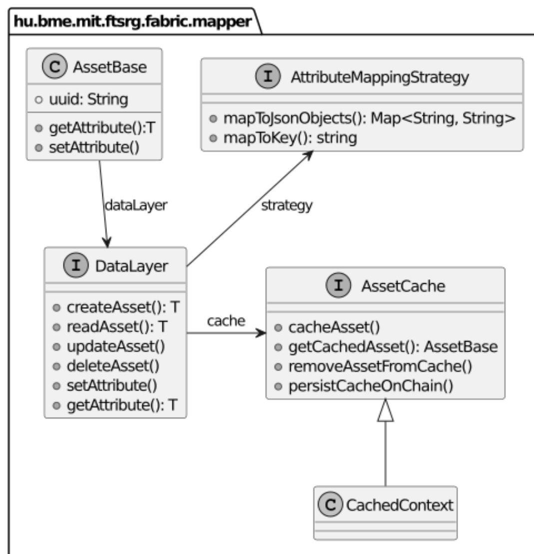

FIGURE 7. Proposed data mapper architecture.

storage) total partitioning method. However, it also has some potential drawbacks.

- Its effectiveness is sensitive to the quality of the apriori transaction data, which is either hypothesized or measured. Thus, it is best suitable when the assumptions about the access patterns are reliable, for example, when a legacy system is being "blockchainified."
- It's hard to define and compare different affinity metrics for attributes to quantify their similarity.
- The resulting partitioning needs to be reevaluated upon changes in the apriori information, e.g., in the workload profile or transaction logic, which can be an expensive operation and may require a maintenance window.

B. A SUPPORTING DATA MAPPER ARCHITECTURE

The two partitioning approaches have been implemented in a prototype Java chaincode SDK. Furthermore, for attribute access-based partitioning, a Python-based Jupyter Notebook has also been created.

The SDK prototype automatically translates storage accesses to the PIM level and vice versa, enabling developers to write succinct, readable, and maintainable code similar to non-Fabric business logic implementations. Entity class attributes can be marked with Java annotations, and the storage-level data model is generated automatically with the configured partitioning approach. Both partitioning approaches have the same development experience; the only difference is in the annotation values that determine the partition of the attribute.

Figure 7 highlights the main part of the SDK's architecture, with a *Data Mapper* pattern at its core and the following key classes:

- `CachedContext`: as per the recommended programming model for Fabric chaincodes, `CachedContext` specializes the `Context` class of the Fabric chaincode SDK. The class instances act as scopes for different transactions. A dedicated instance is passed to each transaction invocation handler/function as a mandatory first parameter, providing ledger-level and custom services. The SDK's custom instances also act as a caching service (`AssetCache` below) between the chaincode and the ledger, extending the base chaincode API with (the by-default unsupported) chaincode-level "read-your-write" consistency capability.
- `AssetBase`: the base class that every business-level entity must inherit to gain partitioning capability. The base class provides generalized attribute-level data manipulation and maintains connections (through injected dependencies) with the other core services of the SDK.
- `DataLayer`: provides CRUD (create, read, update, delete) operations and supports attribute-level manipulation of assets. The layer is also aware of the partitioning capabilities of the entities and retrieves partitioning information about the entities through an injected strategy implementation.
- `AttributeMappingStrategy`: following the strategy design pattern, the concrete implementations of the class decide how to map attributes of assets to key-value entries on the ledger. This class is the core component of the data mapper pattern.
- `AssetCache`: interface for extending the ledger's CRUD operations with caching capabilities. The interface is currently implemented by the `CachedContext` class in the current prototype. However, it could be applied directly to the official Fabric chaincode SDK's data access layer via the *Decorator* pattern.

The caching mechanism of the SDK ensures that race conditions do not occur concerning the independent updates of the same asset on the chaincode. Whenever an asset needs to be retrieved from the ledger, the local cache is consulted first (Figure 8). If the asset instance is already loaded, the same deserialized instance is returned to the client, i.e., the business logic in the chaincode that interacts with the SDK.

The SDK provides automatic life-cycle management for the loaded and manipulated assets (Figure 9). The cache tracks the retrieved assets and their changes, i.e., whether any attributes have been updated. The corresponding partitions are only saved to the ledger if their attributes change. The "save" happens only upon the success of the TX's endorsement, eliminating meaningless write requests to the ledger.

C. EMPIRICAL VALIDATION

The prototype SDK implementation was evaluated using a micro-benchmark to compare the different partitioning approaches and the "unpartitioned" baseline.

1) MICRO-BENCHMARK AND WORKLOAD DEFINITIONS

The data model under test is a personal account definition with multiple subaccounts (similarly to Figure 5), referred

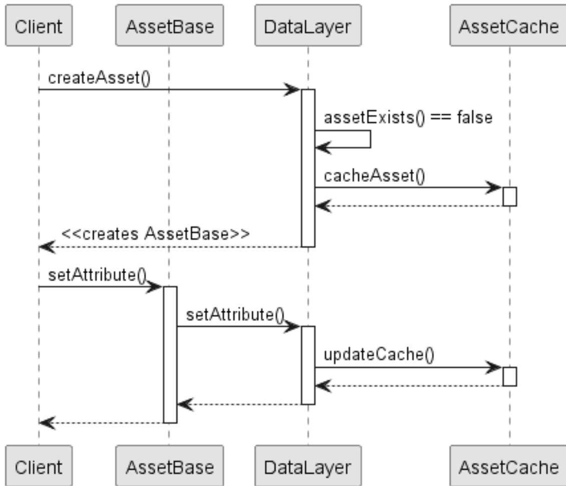

FIGURE 8. The call sequence for creating an asset.

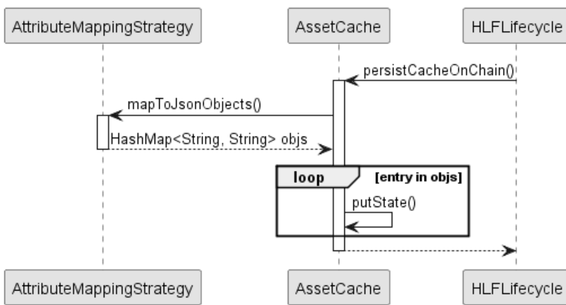

FIGURE 9. The call sequence for persisting an asset.

to as *pockets*. The business logic (i.e., the chaincode) consists of multiple transaction types/functions that modify a different number of pockets, simulating the attribute access-based complexity of general business logic. The transaction functions update one or more pockets of a single account, i.e., they simulate one end of a transfer operation (to keep the randomness of the workload and its evaluation manageable).

Each benchmark run begins with an *initialization* phase, where the accounts are created sequentially. Each account has three subaccounts (pockets). The *main* phase consists of 10000 transactions, where accounts are chosen to be updated randomly – with a constant workload rate within a given benchmark run.

We defined three distinct workload profiles for the chaincode based on the access pattern of the transaction functions:

- **1pockets (1P)**: a transaction updates only a single, fixed subaccount.
- **2pockets (2P)**: a transaction updates two subaccount balances together, or only the third one. Technically, this is a choice between two chaincode methods.
- **3pockets (3P)**: a transaction updates any of the three subaccounts.

TABLE 5. The control variables of the evaluation campaign.

Control Variable	Possible Values
Main account count	200, 2000, 20000
Workload rate (TPS)	30, 60, 90, 120
Workload profile	1P, 2P, 3P
Applied partitioning	Unpartitioned, Affinity-based, Total

2) CAMPAIGN VARIABLES AND TARGET METRICS

Table 5 outlines our experiment control variables and their possible values for the individual benchmark runs. We measured all variable-value combinations in our measurement campaign.

The ranges for the account numbers and workload rates collectively cover the spectrum of high-intensity, hot-key, and low-intensity key access scenarios (i.e., from a few accounts with high load rates to a lot of accounts with low load rates). Different workload profiles simulated how transaction functions can update one or more attributes of a single ledger entity.

In the case of the affinity-based approach, the workload profile *3P* was not tested, as the calculated partitioning scheme matched that of the total partitioning approach.

For evaluation, we used the following metrics.

- **Transaction failure rate** is the ratio of the number of failed transactions to the total number of transactions. In our case, failures mainly occurred due to MVCC conflicts since the chaincode did not employ business logic-level assertions that could fail a TX endorsement. Thus, the transaction failure rate directly translates to the MVCC conflict rate.
- **End-to-end latency** is the total client-side elapsed time from the submission of a transaction proposal to receiving transaction validation notification.
- **Mean read set size** is the combined size of ledger entries (counting both the keys and values in bytes) read during the endorsement of the transaction. Note that even though the consensus process only operates on the versions of the keys (and not their values) for read operations, this is still an important metric, as data retrieval can significantly contribute to the latency and resource use of the endorsement and block validation consensus steps.
- **Mean write set size** is the combined size of ledger entries (counting both the keys and values in bytes) updated during the endorsement of the transaction.

3) MEASUREMENT ENVIRONMENT

The benchmark runs were performed in our university's private cloud on 7 QUEMU-based virtual machines (VM) configured with 8 vCPUs and 16 GB of RAM each. The operating system was Ubuntu 18.04 LTS with Docker 20.10, and the containerized components (Fabric, monitoring components, workload generators, etc.) were all orchestrated by a Docker Swarm network spanning all VMs.

A Hyperledger Fabric v2.22 network was used as the system under test, consisting of 3 nodes: 2 peer nodes (1-1 node per organization) and a single ordering service node. All nodes were deployed on separate VMs to limit performance interferences.

A Fabric batch timeout of 750 ms and a max transaction count per block of 80 were chosen as reasonable middle grounds between throughput and latency. GoLevelDB was used as the state, index, and history database in the peers. All remaining configuration options were left at their default values.

The Hyperledger Caliper[5] blockchain performance benchmarking tool was used for workload generation due to its flexibility and easy-to-use distributed workload generation feature. The benchmark runs utilized four Caliper worker services and an orchestrator service, sharing a single VM with ample capacity.

For instrumentation, the following tools were deployed.

- Hyperledger Explorer (now Blockchain Explorer[6]) was installed on a separate VM to gather data about the read and write sets of the transactions.
- A custom build of Hyperledger Caliper was utilized to collect data about the latency and validity of individual transactions.
- The SDK was instrumented with the help of a logging library; the logs were collected with Logspout,[7] mined with Logstash,[8] and stored in an Elasticsearch[9] instance.
- The utilization of system resources was monitored on all VMs with cmonitor.[10]

4) EVALUATION RESULTS

Overall, we can state that applying any partitioning technique reduces the ratio of MVCC conflicts when the access of attributes is sparse enough (i.e., the 2P and 3P workload profiles) and not focused on a single hot attribute (i.e., the 1P workload profile).

Figure 10 to 12 present the pair-wise comparison of partitioning techniques and the unpartitioned case. The axes represent the ratio of transactions that failed due to MVCC conflicts in a benchmark run; a point in the figures represents a pair of benchmark runs that differ only in the applied partitioning techniques, and their other control variables have the same value. The lines represent the theoretical maximum of improvement for the three workload profiles, as suggested by the hot key theorem in [16] (the dotted line representing $y = 3x$, the dashed line $y = 2x$ and the solid line $y = x$).

Figure 10 and 11 show that both partitioning techniques scale according to the hot key theorem. The difference is especially noticeable for the scenarios that have a high conflict ratio in the unpartitioned case. Figure 12 demonstrates that – for this particular use case – both partitioning methods prevented conflicts with the same effectiveness.

Figure 13 presents the box plots of the end-to-end latencies of transactions for each technique. While there were some outliers, further examination revealed that most of these values occurred during the initialization phase. Thus, all approaches exhibited the same performance characteristics during the main part of the benchmark run, leading to the

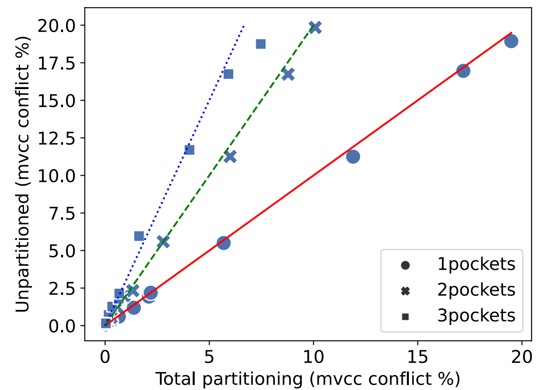

FIGURE 10. Comparison of MVCC conflict percentages: total partitioning vs unpartitioned.

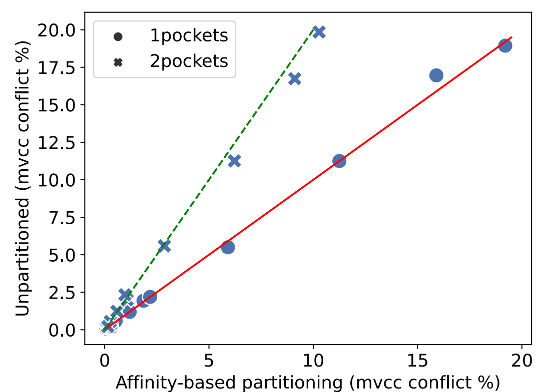

FIGURE 11. Comparison of MVCC conflict percentages: affinity-based partitioning vs unpartitioned.

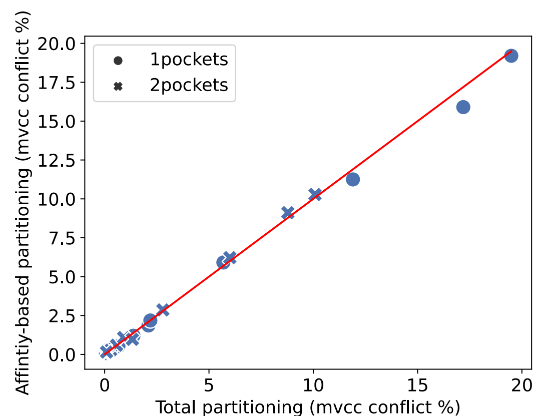

FIGURE 12. Comparison of MVCC conflict percentages: total vs affinity-based partitioning.

[5]https://github.com/hyperledger/caliper
[6]https://github.com/hyperledger-labs/blockchain-explorer
[7]https://github.com/gliderlabs/logspout
[8]https://www.elastic.co/logstash/
[9]https://www.elastic.co/
[10]https://github.com/f18m/cmonitor

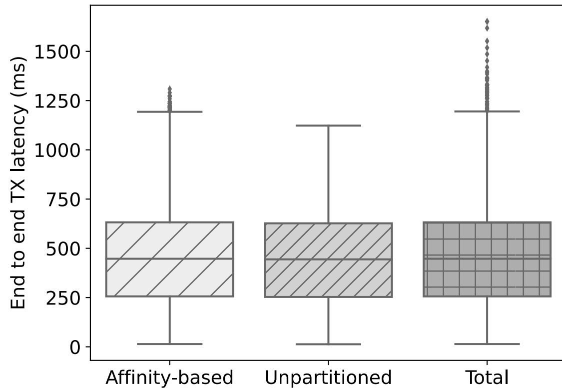

FIGURE 13. End-to-end transaction latencies.

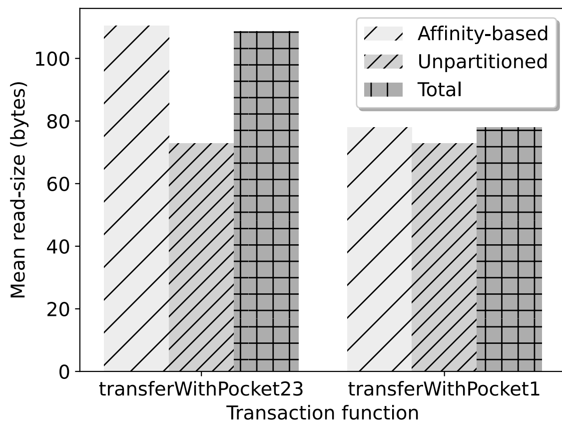

FIGURE 14. Mean size of chaincode-read data.

exploratory data analysis-based hypothesis that the runtime performance overhead of the partitioning techniques is insignificant. Note that the large latency variance is entirely natural for Fabric – the 750 ms "latching" of block formation in itself adds a significant minimum spread.

Figure 14 presents the mean read set sizes produced by the different chaincode functions corresponding to the three subaccount workload profiles. `transferWithPocket23` implements double-pocket transfer for the 2P workload profile; `transferWithPocket1` implements the single-pocket transactions.

The partitioning methods introduced a noticeable read set size overhead when reading multiple attributes. This effect is due to the additional metadata stored and read with the partitions.

Figure 15 presents the mean write set sizes. Metadata introduces overheads when initially creating the accounts with their attributes and partition metadata. However, once an account is created, subsequent updates to the partitions can be smaller than with the unpartitioned approach. Accordingly, the partitioning techniques can lead to reduced I/O load on the blockchain storage while committing the transactions in the validation phase of the consensus protocol.

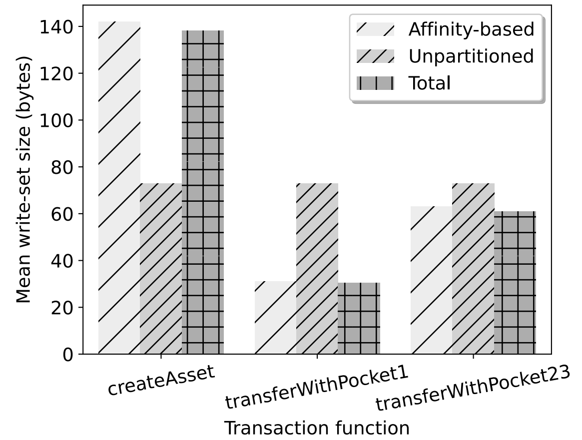

FIGURE 15. Mean size of chaincode-written data.

D. TOWARDS PRODUCTION APPLICATIONS

We do not see any significant threat to the basic validity of our data mapper-based entity attribute partitioning approach; although it provides a different – in our view, preferable – programming model for the business logic part of the chaincode, it does not modify the platform at all, and we have demonstrated on synthetic examples that it can be effective under the right circumstances. Total attribute partitioning is straightforward and robust, although potentially wasteful in storage – however, the storage overhead is easy to assess on an application-by-application basis.

However, how robust affinity-based partitioning is for typical production applications remains to be determined. Similarly, while valid, attribute partitioning can be *ineffective* – ideally, the typical cases when a PIM-level intervention is needed should be identified.

These are significant challenges. Open and production-grade chaincodes are practically nonexistent (primarily due to the consortial and bespoke nature of Fabric-based solutions), and chaincode benchmark standardization efforts are still in their infancy. The authors are currently working on porting their "blockchainified" TPC-C performance benchmark implementation [8] to Java, the chaincode language utilized by the SDK prototype. The workload specification of TPC-C enables the controlled scaling of the MVCC conflict ratio in the system while operating with a complex data model.

The implementation efficiency of the SDK prototype also has room for improvement. More precisely, the observed read and write set overheads might be substantially reduced by using a more efficient encoding of partitions or even the attribute data itself (for example, using Protobufs instead of JSON).

VI. SUMMARY AND FUTURE WORK

In this paper, with the help of a novel taxonomy, we presented our findings on the gaps in the current Hyperledger Fabric MVCC conflict control literature. We drew attention to

the importance of designing for MVCC conflict prevention during system engineering and showed that it requires following model-driven engineering principles.

In that context, we proposed data model partitioning techniques to facilitate conflict prevention as a previously overlooked avenue of MVCC conflict control. The algorithms we propose, total partitioning and affinity-based partitioning, prioritize different aspects and are suitable for a wide range of use cases. We implemented these approaches as part of an SDK prototype which acts as a data mapper chaincode layer and evaluated them on micro-benchmark scenarios.

Our results show that the affinity-based approach reduces the number of database accesses and matches the total partitioning approach in conflict mitigation in the case of well-known access patterns, but it may not necessarily reduce storage use.

We believe our proposed techniques hold promise for further research and development. We plan to investigate more efficient storage methods, adding domain modeling to the framework and prototyping a full-fledged MDE process. More representative validation on a TPC-C implementation for Hyperledger Fabric is also planned.

REFERENCES

[1] G. Wood. *Ethereum: A Secure Decentralised Generalised Transaction Ledger*. Accessed: Nov. 28, 2023. [Online]. Available: https://gavwood.com/paper.pdf

[2] D. Yaga, P. Mell, N. Roby, and K. Scarfone, "Blockchain technology overview," 2019, *arXiv:1906.11078*.

[3] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, and S. Muralidharan, "Hyperledger Fabric: A distributed operating system for permissioned blockchains," in *Proc. 13th EuroSys Conf.*, 2018, pp. 1–15, doi: 10.1145/3190508.3190538.

[4] P. A. Bernstein and N. Goodman, "Multiversion concurrency control—Theory and algorithms," *ACM Trans. Database Syst.*, vol. 8, no. 4, pp. 465–483, 1983, doi: 10.1145/319996.319998.

[5] M. Debreczeni. *Data Model Driven Goodput Optimization for Execute-Order-Validate Blockchains*. Accessed: Nov. 28, 2023. [Online]. Available: https://tdk.bme.hu/VIK/ViewPaper/Alkalmazasi-szintu-ateresztokepesseg

[6] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. USENIX Annu. Tech. Conf.*, 2014, pp. 305–319. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[7] J. A. Chacko, R. Mayer, and H.-A. Jacobsen, "Why do my blockchain transactions fail? A study of Hyperledger Fabric," in *Proc. Int. Conf. Manag. Data*, 2021, pp. 221–234, doi: 10.1145/3448016.3452823.

[8] A. Klenik and I. Kocsis, "Porting a benchmark with a classic workload to blockchain: TPC-C on Hyperledger Fabric," in *Proc. 37th ACM/SIGAPP Symp. Appl. Comput.*, Apr. 2022, pp. 290–298, doi: 10.1145/3477314.3507006.

[9] J. A. Chacko, R. Mayer, and H.-A. Jacobsen, "How to optimize my blockchain? A multi-level recommendation approach," *Proc. ACM Manag. Data*, vol. 1, no. 1, pp. 1–27, May 2023, doi: 10.1145/3588704.

[10] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich, "Blurring the lines between blockchains and database systems: The case of Hyperledger Fabric," in *Proc. Int. Conf. Manag. Data*, 2019, pp. 105–122, doi: 10.1145/3299869.3319883.

[11] M. Kim, S. Lee, C. Park, and J. Lee, "Age of information analysis in Hyperledger Fabric blockchain-enabled monitoring networks," in *Proc. IEEE Int. Conf. Commun.*, Jun. 2021, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9500864

[12] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable Secure Comput.*, vol. 1, no. 1, pp. 11–33, Jan. 2004.

[13] R. Hanmer, *Patterns for Fault Tolerant Software*. Hoboken, NJ, USA: Wiley, 2013.

[14] P. Nasirifard, R. Mayer, and H.-A. Jacobsen, "OrderlessChain: Do permissioned blockchains need total global order of transactions?" 2023, *arXiv:2210.01477*.

[15] A. Alzubaidi, K. Mitra, and E. Solaiman, "Smart contract design considerations for SLA compliance assessment in the context of IoT," in *Proc. IEEE Int. Conf. Smart Internet Things (SmartIoT)*, Aug. 2021, pp. 74–81. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9556177

[16] C. Gorenflo, S. Lee, L. Golab, and S. Keshav, "FastFabric: Scaling Hyperledger Fabric to 20 000 transactions per second," *Int. J. Netw. Manag.*, vol. 30, no. 5, p. e2099, Sep. 2020, doi: 10.1002/nem.2099.

[17] Z. István, A. Sorniotti, and M. Vukolic, "StreamChain: Do blockchains need blocks?" in *Proc. 2nd Workshop Scalable Resilient Infrastructures for Distrib. Ledgers*, Dec. 2018, pp. 1–6 doi: 10.1145/3284764.3284765.

[18] P. Nasirifard, R. Mayer, and H.-A. Jacobsen, "FabricCRDT: A conflict-free replicated datatypes approach to permissioned blockchains," in *Proc. 20th Int. Middleware Conf.*, Dec. 2019, pp. 110–122 doi: 10.1145/3361525.3361540.

[19] L. Xu, W. Chen, Z. Li, J. Xu, A. Liu, and L. Zhao, "Solutions for concurrency conflict problem on Hyperledger Fabric," *World Wide Web*, vol. 24, no. 1, pp. 463–482, Jan. 2021, doi: 10.1007/s11280-020-00851-6.

[20] S. Zhang, E. Zhou, B. Pi, J. Sun, K. Yamashita, and Y. Nomura, "A solution for the risk of non-deterministic transactions in Hyperledger Fabric," in *Proc. IEEE Int. Conf. Blockchain Cryptocurrency*, May 2019, pp. 253–261. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8751453

[21] Q. Sun, Y. Yuan, T. Guo, and L. Chen, "A trusted solution to Hyperledger Fabric reordering problem," in *Proc. 8th Int. Conf. Dependable Syst. Their Appl. (DSA)*, Aug. 2021, pp. 202–207. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9622776

[22] P. Ruan, D. Loghin, Q.-T. Ta, M. Zhang, G. Chen, and B. C. Ooi, "A transactional perspective on execute-order-validate blockchains," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, Jun. 2020, pp. 543–557 doi: 10.1145/3318464.3389693.

[23] H. Wu, H. Liu, and J. Li, "FabricETP: A high-throughput blockchain optimization solution for resolving concurrent conflicting transactions," *Peer-Peer Netw. Appl.*, vol. 16, no. 2, pp. 858–875, Mar. 2023, doi: 10.1007/s12083-022-01401-9.

[24] X. Xu, X. Wang, Z. Li, H. Yu, G. Sun, S. Maharjan, and Y. Zhang, "Mitigating conflicting transactions in Hyperledger Fabric-permissioned blockchain for delay-sensitive IoT applications," *IEEE Internet Things J.*, vol. 8, no. 13, pp. 10596–10607, Jul. 2021. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9317791

[25] C. Gorenflo, L. Golab, and S. Keshav, "XOX Fabric: A hybrid approach to blockchain transaction execution," in *Proc. IEEE Int. Conf. Blockchain Cryptocurrency*, May 2020, pp. 1–9. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9169478

[26] H. Trabelsi and K. Zhang, "Early detection for multiversion concurrency control conflicts in Hyperledger Fabric," 2023, *arXiv:2301.06181*.

[27] C. Wu, B. Mehta, M. J. Amiri, R. Marcus, and B. T. Loo, "AdaChain: A learned adaptive blockchain," *Proc. VLDB Endowment*, vol. 16, no. 8, pp. 2033–2046, Apr. 2023, doi: 10.14778/3594512.3594531.

[28] M. Li, Y. Wang, S. Ma, C. Liu, D. Huo, Y. Wang, and Z. Xu, "Auto-tuning with reinforcement learning for permissioned blockchain systems," *Proc. VLDB Endowment*, vol. 16, no. 5, pp. 1000–1012, Jan. 2023, doi: 10.14778/3579075.3579076.

[29] Y. Lu, C. Liu, M. Zhao, X. Duo, P. Xu, Z. Zhou, and X. Feng, "FSC: A fast smart contract transaction execution approach via read-write static analysis," Tech. Rep., 2023. [Online]. Available: https://www.authorea.com/doi/full/10.22541/au.167285898.83759504

[30] L. Hang, B. Kim, and D. Kim, "A transaction traffic control approach based on fuzzy logic to improve Hyperledger Fabric performance," *Wireless Commun. Mobile Comput.*, vol. 2022, pp. 1–19, Mar. 2022. [Online]. Available: https://www.hindawi.com/journals/wcmc/2022/2032165/

[31] F. Jamil, S. Ahmad, T. K. Whangbo, A. Muthanna, and D.-H. Kim, "Improving blockchain performance in clinical trials using intelligent optimal transaction traffic control mechanism in smart healthcare applications," *Comput. Ind. Eng.*, vol. 170, Aug. 2022, Art. no. 108327. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0360835222003801

[32] P. Zheng, Z. Jiang, J. Wu, and Z. Zheng, "Blockchain-based decentralized application: A survey," *IEEE Open J. Comput. Soc.*, vol. 4, pp. 121–133, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10068327

[33] Object Management Group. *Overview and Guide to OMG's Architecture*. Accessed: Nov. 28, 2023. [Online]. Available: https://www.omg.org/cgi-bin/doc.cgi?omg/03-06-01

[34] A. Rodrigues da Silva, "Model-driven engineering: A survey supported by the unified conceptual model," *Comput. Lang., Syst. Struct.*, vol. 43, pp. 139–155, Oct. 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1477842415000408

[35] S. Curty, F. Härer, and H.-G. Fill, "Design of blockchain-based applications using model-driven engineering and low-code/no-code platforms: A structured literature review," *Softw. Syst. Model.*, vol. 22, no. 6, pp. 1857–1895, Dec. 2023, doi: 10.1007/s10270-023-01109-1.

[36] M. Hamdaqa, L. A. P. Met, and I. Qasse, "IContractML 2.0: A domain-specific language for modeling and deploying smart contracts onto multiple blockchain platforms," *Inf. Softw. Technol.*, vol. 144, Apr. 2022, Art. no. 106762. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584921002081

[37] M. Hamdaqa, L. A. P. Metz, and I. Qasse, "IContractML: A domain-specific language for modeling and deploying smart contracts onto multiple blockchain platforms," in *Proc. 12th Syst. Anal. Model. Conf.*, Oct. 2020, pp. 34–43 doi: 10.1145/3419804.3421454.

[38] P. Fraternali, S. L. H. Gonzalez, M. Frigerio, and M. Righetti, "Model-driven development of distributed ledger applications," in *Database Systems for Advanced Applications. DASFAA 2022 International Workshops* (Lecture Notes in Computer Science), U. K. Rage, V. Goyal, and P. K. Reddy, Eds. Cham, Switzerland: Springer, 2022, pp. 104–119.

[39] D. Aveiro, L. Abreu, D. Pinto, and V. Freitas, "DEMO models based automatic smart contract generation: A case in logistics using Hyperledger," in *Information Systems Development, Organizational Aspects and Societal Trends*. Lisbon, Portugal: Instituto Superior Técnico, 2023. [Online]. Available: https://aisel.aisnet.org/isd2014/proceedings2023/modelling/3

[40] M. Jurgelaitis, L. Ceponiene, K. Butkus, R. Butkiene, and V. Drungilas, "MDA-based approach for blockchain smart contract development," *Appl. Sci.*, vol. 13, no. 1, p. 487, Dec. 2022. [Online]. Available: https://www.mdpi.com/2076-3417/13/1/487

[41] M. Jurgelaitis, V. Drungilas, L. Eponien, E. Vaiiukynas, R. Butkien, and J. Ceponis, "Smart contract code generation from platform specific model for Hyperledger go," in *Trends and Applications in Information Systems and Technologies* (Advances in Intelligent Systems and Computing), A. Rocha, H. Adeli, G. Dzemyda, F. Moreira, and A. M. R. Correia, Eds. Cham, Switzerland: Springer, 2021, pp. 63–73.

[42] V. A. de Sousa and C. Burnay, "MDE4BBIS: A framework to incorporate model-driven engineering in the development of blockchain-based information systems," in *Proc. 3rd Int. Conf. Blockchain Comput. Appl. (BCCA)*, Nov. 2021, pp. 195–200. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9657015

[43] E. Babkin and N. Komleva, "Model-driven liaison of organization modeling approaches and blockchain platforms," in *Advances in Enterprise Engineering XIII* (Lecture Notes in Business Information Processing), D. Aveiro, G. Guizzardi, and J. Borbinha, Eds. Cham, Switzerland: Springer, 2020, pp. 167–186.

[44] J. Ladleif, C. Friedow, and M. Weske, "An architecture for multi-chain business process choreographies," in *Business Information Systems* (Lecture Notes in Business Information Processing), W. Abramowicz and G. Klein, Eds. Cham, Switzerland: Springer, 2020, pp. 184–196.

[45] F. Corradini, A. Marcelletti, A. Morichetta, A. Polini, B. Re, E. Scala, and F. Tiezzi, "Model-driven engineering for multi-party business processes on multiple blockchains," *Blockchain, Res. Appl.*, vol. 2, no. 3, Sep. 2021, Art. no. 100018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2096720921000130

[46] P. Bodorik, C. G. Liu, and D. Jutla, "TABS: Transforming automatically BPMN models into blockchain smart contracts," *Blockchain: Res. Appl.*, vol. 4, no. 1, Mar. 2023, Art. no. 100115. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2096720922000562

[47] H. Nakamura, K. Miyamoto, and M. Kudo, "Inter-organizational business processes managed by blockchain," in *Web Information Systems Engineering—WISE* (Lecture Notes in Computer Science), H. Hacid, W. Cellary, H. Wang, H.-Y. Paik, and R. Zhou, Eds. Cham, Switzerland: Springer, 2018, pp. 3–17.

[48] T. Astigarraga, X. Chen, Y. Chen, J. Gu, R. Hull, L. Jiao, Y. Li, and P. Novotny, "Empowering business-level blockchain users with a rules framework for smart contracts," in *Service-Oriented Computing* (Lecture Notes in Computer Science), C. Pahl, M. Vukovic, J. Yin, and Q. Yu, Eds. Cham, Switzerland: Springer, 2018, pp. 111–128.

[49] O. Choudhury, N. Rudolph, I. Sylla, N. Fairoza, and A. Das, "Auto-generation of smart contracts from domain-specific ontologies and semantic rules," in *Proc. IEEE Int. Conf. Internet Things (iThings), IEEE Green Comput. Commun. (GreenCom), IEEE Cyber, Phys. Social Comput. (CPSCom), IEEE Smart Data (SmartData)*, Jul. 2018, pp. 963–970.

[50] A. Rasti, D. Amyot, A. Parvizimosaed, M. Roveri, L. Logrippo, A. A. Anda, and J. Mylopoulos, "Symboleo2SC: From legal contract specifications to smart contracts," in *Proc. 25th Int. Conf. Model Driven Eng. Lang. Syst.*, 2022, pp. 300–310, doi: 10.1145/3550355.3552407.

[51] N. Sato, T. Tateishi, and S. Amano, "Formal requirement enforcement on smart contracts based on linear dynamic logic," in *Proc. IEEE Int. Conf. Internet Things (iThings), IEEE Green Comput. Commun. (GreenCom), IEEE Cyber, Phys. Social Comput. (CPSCom), IEEE Smart Data (SmartData)*, Jul. 2018, pp. 945–954. [Online]. Available: https://ieeexplore.ieee.org/document/8726750

[52] S. Navathe, S. Ceri, G. Wiederhold, and J. Dou, "Vertical partitioning algorithms for database design," *ACM Trans. Database Syst.*, vol. 9, no. 4, pp. 680–710, Dec. 1984, doi: 10.1145/1994.2209.

[53] W. T. McCormick, P. J. Schweitzer, and T. W. White, "Problem decomposition and data reorganization by a clustering technique," *Oper. Res.*, vol. 20, no. 5, pp. 993–1009, Oct. 1972, doi: 10.1287/opre.20.5.993.

MÁTÉ DEBRECZENI received the B.Sc. degree in computer engineering from the Budapest University of Technology and Economics (BME), in 2022, where he is currently pursuing the M.Sc. degree in computer engineering, with a focus on cybersecurity specialization. He is also a Junior Analyst with the Central Bank of Hungary (MNB). His main professional research interests include the design and optimization of data-intensive applications, including the empirical analysis of distributed ledger deployments.

ATTILA KLENIK received the M.Sc. degree in computer engineering and the Ph.D. degree from the Budapest University of Technology and Economics (BME), in 2022. He is currently a Senior Researcher with the Critical Systems Research Group, Department of Measurement and Information Systems, and the Core Developer of the hyperledger caliper blockchain benchmarking tool. His area of expertise is the distributed tracing and measurement-based performance evaluation of distributed systems, with a special focus on consortial blockchain technologies, such as Hyperledger Fabric.

IMRE KOCSIS received the M.Sc. degree in computer engineering and the Ph.D. degree from the Budapest University of Technology and Economics (BME), in 2019. Currently, he is a Senior Lecturer and a Leading Blockchain Researcher with the Critical Systems Research Group, Department of Measurement and Information Systems, BME. He leads the activities of the group in conjunction with the Hyperledger Foundation and the university's participation in the European Blockchain Services Infrastructure (EBSI) networks. His professional research interests include the requirement-based design and dependability assurance of cross-organizational blockchains, industrial use cases, inter-blockchain integration, and the blockchain support of Central Bank Digital Currencies.