Operation-based Collaborative Data Sharing for Distributed Systems

Masato Takeichi

takeichi@acm.org

November 26, 2021

Abstract. Collaborative Data Sharing raises a fundamental issue in distributed systems. Several strategies have been proposed for making shared data consistent between peers in such a way that the shared part of their local data become equal.

Most of the proposals rely on *state-based* semantics. But this suffers from a lack of descriptiveness in conflict-free features of synchronization required for flexible network connections. Recent applications tend to use non-permanent connection with mobile devices or allow temporary breakaways from the system, for example.

To settle ourselves in conflict-free data sharing, we propose a novel scheme *Operation-based Collaborative Data Sharing* that enables conflict-free strategies for synchronization based on operational semantics.

1 Introduction

Each site, or peer of distributed systems has its exclusive property of the contents and the policy for data sharing. For collaborative work between peers, the peer expects partner peers to receive some of its data and asks them for returning the updated data, or it asks partner peers to provide their data for use with its local data.

This kind of data sharing is common in our real-world systems. Although data sharing without updates is simple, collaborative data sharing with the update propagation of shared data poses significant problems due to concurrent updates of different instances of the same data. Which updates should be allowed or how the update should be propagated to all the related peers are the typical issues to be solved.

We have been discussing "What should be shared" in collaborative data sharing, but not so much talking about "How should be shared".

Concerning the "what", a seminal work on *Collaborative Data Sharing* [8, 9] brought several issues upon the specification of data to be shared. An approach based on the view-updating technique with *Bidirectional Transformation* [3, 2, 4, 6, 5] has been proved promissing. Among others, the *Dejima* 1.0 architecture [7, 1] and the *BCDS Agent* [11] based on Bidirectional Transformation reveal the effectiveness of using Bidirectional Transformation for peers to control the accessibility of local data.

The basic scheme of these ideas is based on *state-based* semantics. That is, data to be shared is compared with and moved to and from between peers. Although this is straightforward in a sense, there may be several problems; the size of messages for data exchange tends to grow, and possible conflicts occur due to concurrent updates.

Looking from the othr side, how to share data between peers is very similar to how to synchronize distributed replicas to be the same. They are almost equivalent except original intentions. And our problem to be solved is how to synchronize distributed replicas in serverless distributed systems.

We have various kind of *Conflict-free Replicated Data Types* (CRDTs) [10]. The CRDT approach restricts available operations acted on replicated data; the *Grow-Only-Set* (G-Set) CRDT allows only the insertion operation on the set data, for example.

In this paper, we will explore a novel scheme for collaborative data sharing based on *operation-based* approach. The semantics of collaborative data sharing is redefined using operations performed on peer's local data. And as a natural course, operations are exchanged each other for making effective data sharing between peers.

Our *Operation-based Collaborative Data Sharing* (OCDS) can solve the problem concerning possible conflicts between concurrent operations by conflict-free synchronization for eventual consistency. And this accepts more operations than CRDTs. It is the most remarkable feature of our OCDS compared with CRDTs.

2 Operations and Transformations in Data Sharing

The *Dejima* architecture mentioned in the previous section configures peers with local data called *Base Table* and several additional *Dejima Tables*. The shared data is located both in the Dejima Tables in peers P and Q as illustrated in Fig.1. *Bidirectional Transformation* is employed to convert data between the Base Table and the Dejima Table. It controls what to provide and what to accept for data sharing.

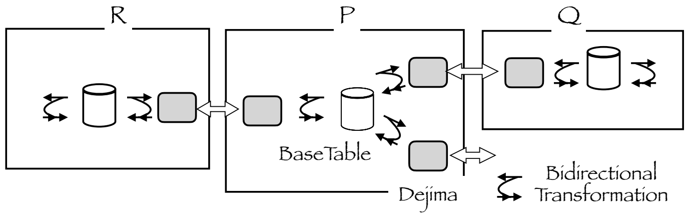


Fig. 1. Dejima Architecture for Collaborative Data Sharing

As described above, the Dejima architecture relies on *state-based* semantics.

We will give another definition of *collaborative data sharing* based on operations on the local data. We assume that the configuration of peers is the same as the Dejima architecture without the Dejima Table which corresponds to the state of shared data.

Updating Operation Peer P has local data D_P of (structured) type $\mathcal{D}_P$ with operations on $\mathcal{D}_P$. The operation takes postfix form $\odot_P p :: \mathcal{D}_P \to \mathcal{D}_P$ where $p \in \cup_{D_P \in \mathcal{D}_\mathcal{P}} D_P$, which maps D_P to $D_P \odot_P p \in \mathcal{D}_P$, i.e., $D_P \mapsto D_P \odot_P p$.

From operational point of view, "$\odot_P p$ updates D_P into $D_P \odot_P p$". Here, p is not necessarily a single element, but may be composed of several elements in $\cup_{D_P \in \mathcal{D}_\mathcal{P}} D_P$. But, for simplicity, we write them as a single p.

Operation $\odot_P p$ is derived from operator $\odot_P$ in P indexed by p. Operator $\odot_P$ stands for a generic symbol for $\oplus$, $\ominus$, $\otimes$, *oslash*, etc. For convenience, we use a postfix identity operation "!" which does not change D_P, i.e., $D_P! = D_P$ for any $D_P \in \mathcal{D}_P$.

Transformation Function In addition to these operations, transformation functions $\langle get^q_P, put^q_P \rangle$, where $get^q_P :: \mathcal{D}_P \to \mathcal{D}$ and $put^q_P :: \mathcal{D} \to \mathcal{D}_P$ for some $\mathcal{D}$.

With a reservation, put^q_P may use $\mathcal{D}_P$ along with $\mathcal{D}$ as $put^q_P :: \mathcal{D}_P \times \mathcal{D} \to \mathcal{D}_P$.

Same for the partner peer Q with D_Q and $\langle get^p_Q, put^p_Q \rangle$, and $\mathcal{D}$ appears in definitions in Q is common to $\mathcal{D}$ in P. Thus,$\mathcal{D}$ "combines" P and Q as a connector for data exchange.

Suffixes for identifying the peer, e.g., P in $\mathcal{D}_P$, D_P, $\odot_P$, ... are omitted when they are clear from the context.

Properties of Transformation Functions Given data $D_P \in \mathcal{D}_P$, $get^q_P(D_P)$ gives some $D \in \mathcal{D}$. And then, Q gets share with data $D_Q = put^p_Q(D) \in \mathcal{D}_Q$ which corresponds to D_P in P. And reciprocally, P shares $D'_P = put^q_P(D') = put^q_P(get^p_Q(D'_Q)) \in \mathcal{D}_P$ which corresponds to D'_Q in Q.

Intuitively, we may understand that part of D_P and part of D_Q are shared each other.

If it happened to be $D = D'$, it is natural to assume that $D_P = D'_P$ and $D_Q = D'_Q$ hold with the above equalities, so

$$(put^q_P \cdot get^p_Q) \cdot (put^p_Q \cdot get^q_P) = (put^p_Q \cdot get^q_P) \cdot (put^q_P \cdot get^p_Q) = id$$

hold. Then, what should we require for *get* and *put* in each peer?

Considering that $\langle get^q_P, put^q_P \rangle$ and $\langle get^p_Q, put^p_Q \rangle$ are prepared independently in P and Q, it is reasonable to ask for

$$put^q_P \cdot get^q_P = get^q_P \cdot put^q_P = id$$
$$put^p_Q \cdot get^p_Q = get^p_Q \cdot put^p_Q = id$$

. This is what we call the "Round-tripping" property of *well-behaved* bidirectional transformation. And we require our *get* and *put* to satisfy this property.

To define well-behaved bidirectional transformation $\langle get^q_P, put^q_P \rangle$, taking $\mathcal{D}_P$ along with D as the domain of put^q_P is of a great help to define well-behaved transformation. From this reason, we sometimes define it as $put^q_P :: \mathcal{D}_P \times \mathcal{D} \to \mathcal{D}_P$.

From our operational viewpoint, this put^q_P updates the current instance D_P of mutable data $\mathcal{D}_P$ with $D \in \mathcal{D}$ to produce a new instance $D'_P \in \mathcal{D}_P$. This is natural and reasonable in that we may use the current data when updating mutable data.

3 Operation-based Collaborative Data Sharing

Local operation $\odot_P p$ causes an effect on elements of structured data D_P at a time, and therefore $D_P \odot_P p$ is a new instance $D'_P \in \mathcal{D}_P$ which is almost the same as D_P except for some different elements. A simple example of $\mathcal{D}_P$ is the set with standard operations "insert an element p" (written as $\cup\{p\}$) and "delete an element p"($\backslash\{p\}$).

As for collaborative data sharing between P and Q, a straightforward method for synchronization would be to exchange D_P and D_Q through D with transformation by *get*s and *Put*s at the gateways of P and Q. This approach is called "state-based", and D is called "Dejima".

Although the state-based approach to collaborative data sharing is most common, it is not suitable for *conflict-free* strategies that aim to do something gradually in P and Q for the shared part of D_P and the part of D_Q to arrive at the same state eventually. The conflict-free approach liberates us from the necessity of global locks for exclusive access to the whole distributed data to avoid conflicts between concurrent updates. This is particularly useful in distributed systems with no coordination by any peers such as P2P-configured or composed of highly independent peers.

While the *Conflict-free Replicated Data Type* (CRDT) restricts operations so that the data in each peer can be easily merged, our conflict-free approach allows a wider class of operations that are common to general data structures. Recently, a novel scheme for *Conflict-free Collaborative Set Sharing* [12] is proposed using operations performed so far instead of directly merging the current data. Although this concentrates on the set data, it can be extended to our data sharing where transformations lie between peers' local data.

3.1 Homomorphic Data Structures for Data Sharing

If $D_Q = put^p_Q(get^q_P(D_P))$ and $D_P = put^q_P(get^p_Q(D_Q))$ hold, we say that "D_P and D_Q are *consistent*" and write this as $D_P \sim D_Q$. In other words, consistent D_P and D_Q have corresponding parts which are shared each other through intermediate data D between them.

Assuming that $D_P \sim D_Q$, then what happens when operation $\odot_P p$ is performed on D_P to produce $D_P \odot_P p$?

– If $get^q_P(D_P \odot_P p)$ gives some D' which is to be transformed next by put^p_Q, and

  - If $put^p_Q(D')$ gives some $D'_Q \in \mathcal{D}_Q$, then $D_P \odot_P p \sim D'_Q$.
  - Otherwise, $D_P \odot_P p$ has no corresponding instance in $\mathcal{D}_Q$.
- Otherwise, $D_P \odot_P p$ has no corresponding instance in $\mathcal{D}_Q$.

Since $\odot_P p$ changes some elements of D_P, we hope that D' and D'_Q also change some element as $D' = D \odot x$ and $D'_Q = D_Q \odot_Q q$ with $\odot x$ and $\odot_Q q$.

In most of our data sharing applications, D_P, $D_Q$and intermediate data D are *homomorphic* each other in that the above conditions are satisfied.

In this respect, our transformation functions *get* and *put* partly provide *homomorphism*. The simplest example would be the case where all the related data structures are sets or SQL tables, etc. In general, these are not necessarily the same but are homomorphic. And we need more about homomorphism on operations for our operation-based data sharing.

Homomorphic Data Structures with Operations Data type $\langle \mathcal{A}, \circledcirc_A \rangle$ is closed with respect to operations $\circledcirc_A a$ for any $a \in \cup_{A\in\mathcal{A}} A$, where operator symbol $\circledcirc_A :: (\mathcal{A}, \cup_{A\in\mathcal{A}} A) \to \mathcal{A}$ represents any operators in $\mathcal{A}$. We simply write here $\circledcirc_A$ for the set of operators in $\mathcal{A}$ and use the same symbol for one of them as a generic operator in an overloaded manner.

The operation $\circledcirc_A a :: \mathcal{A} \to \mathcal{A}$ is postfixed to the operand $A \in \mathcal{A}$ to produces $A' = A \circledcirc_A a \in \mathcal{A}$.

This models the *mutable* state data A with operations $\circledcirc_A a$ on A using some element a.

Definition of Homomorphic Data Types Data types $\langle \mathcal{A}, \circledcirc_A \rangle$ and $\langle \mathcal{B}, \circledcirc_B \rangle$ are *homomorphic* if there exist $h :: \mathcal{A} \to \mathcal{B}$ and overloaded $h :: \circledcirc_A \to \circledcirc_B$ satisfying

$$\forall A \in \mathcal{A}. \exists B \in \mathcal{B}. B = h(A)$$
$$\forall A \in \mathcal{A}. \forall a \in \cup_{A\in\mathcal{A}} A. \exists B \in \mathcal{B}. \exists b \in \cup_{B\in\mathcal{B}}. B \circledcirc_B b = h(A \circledcirc_A a)$$

We assume that every data type $\langle \mathcal{A}, \circledcirc_A \rangle$ has an identity operation "!" which does not affect the state of data. That is, for any $A \in \mathcal{A}$, $A! = A$ holds.

In short, for operation-based collaboration, we require operations to exchange between homomorphic data types so that operation on a peer corresponds to operation on the partner peer.

Examples of Homomorphic Data Types Previous works on state-based data sharing with transformation [7, 1, 11] exclusively deal with SQL databases as local data. Specifically, if the intermediate data D is defined as the view of the SQL table D_P of the local data, it is obvious that D_P and D are homomorphic because D is produced by selection and projection of D_P. However, the Dejima architecture allows so-called SPJU (Select-Project-Join-Union) queries by the SQL's SELECT-FROM-WHERE-UNION construct for the view. And we need more to work on making sure that D_P and D are homomorphic. We leave this for the future.

As a demonstration of the independence of implementation of the local data D_P from the intermediate data D of our operation-based data sharing, consider the case that D is a set, i.e., no duplicates in aggregation, and D_P implements set by the binary search tree. In this case, we easily give a homomorphism mapping from D_P to D. Or, it should be grounded in the data abstraction mechanism.

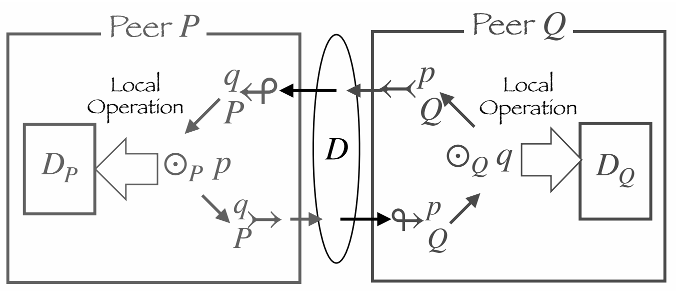
As for the relationship of homomorphic data types with state machines, see the Appendix.

3.2 Transformation of Operations

For homomorphic data structures $\langle \mathcal{D}_P, \odot_P \rangle$, $\langle \mathcal{D}, \odot \rangle$ and $\langle \mathcal{D}_Q, \odot_Q \rangle$, operations are transformed according to the hompmorphisms by *get* and *put*. We write

$\odot_P \overset{q}{\underset{P}{\rightarrowtail}} \odot x$, if $get^q_P(D_P \odot_P p)$ gives $D \odot x$.
$\odot x \overset{p}{\underset{Q}{\hookrightarrow}} \odot_Q q$, if $put^p_Q(D \odot x)$ gives $D_Q \odot_Q q$

Our Operation-based Collaborative Data Sharing wholly sends and receives operations instead of data as shown in the diagram of Fig.2.



Peer P
Peer Q
Local Operation
Local Operation
$\overset{q}{\underset{P}{\hookleftarrow}}$
$\overset{p}{\underset{Q}{\leftarrowtail}}$
D_P
$\odot_P\ p$
D
$\odot_Q\ q$
D_Q
$\overset{q}{\underset{P}{\rightarrowtail}}$
$\overset{p}{\underset{Q}{\hookrightarrow}}$

Fig. 2. Operation-based Collaborative Data Sharing

In this way, peers of our Collaborative Data Sharing communicate updating operations to and from each other with necessary transformation at the gateway of the peer.

4 Architecture for Collaborative Data Sharing

Peers of our Collaborative Data Sharing run concurrently and they transmit their updates asynchronously to and from each other. Thus, The *payload* of the communication message between peers is the updating operation from a peer to its partner peer.

The peer as the *client* sends local operations to the partner peers. And as the *server* receives remote operations from the partners and then perform necessary operations to reflect them on the local data.

In these processes, each peer works as follows. Peer P asynchronously receives local operations from the user and remote operations from the partner peers. These operations are to be performed on D_P and are stored in the queue for serialized access to D_P.

So far, we explained our scheme solely with peers P and Q. But, in general, each peer P has multiple peers connected in the system. For P to do with all the partmer peers, we need to clarify how P should do.

In peer P, every update on data D_P is propagated to all the partner peers $K = \cdots, Q, \cdots$ through the outgoing communication ports prepared for each peer K after it is transformed by ${}^{k}_{P}\!\rightarrowtail$.

And remote operations from peers are received asynchronously from the partner peers $K = \cdots, Q, \cdots$ through the incoming ports each prepared for the peer K and transformed by $\looparrowright^{k}_{P}$.

As the local and remote operations arrive asynchronously, the peer needs to provide queues for them to perform the operations on the local data.

We call the implementation of the peer as described above by the name "OCDS Agent".

The OCDS Agent is developed to achieve conflict-free synchronization of the local data using internal queues for serialization of asynchronous access to the local data and asynchronous transmission of operations as illustrated in Fig.3.

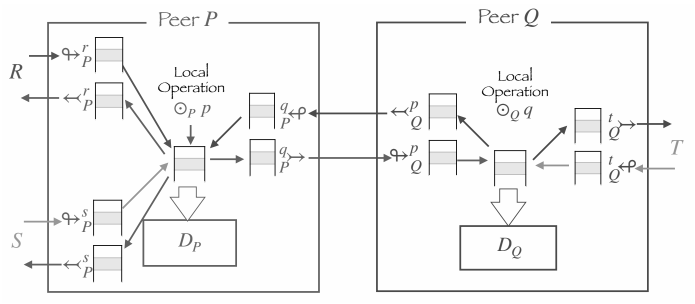


Fig. 3. OCDS Agent for Operation-based Collaborative Data Sharing

5 An Example of Operation-based Collaborative Data Sharing

Sharing Double and Triple Numbers Let D_P be a set of integers with operations "insert an element p" $(\cup\{p\})$ and "delete an element p"$(\backslash\{p\})$.

Bidirectional transformation defied in P is

$$get^p_P(D_P) = \{p \mid p\%2 = 0,\ p \in D_P\}$$
$$put^q_P(D_P, D) = D_P \setminus get^q_P(D_P) \cup \{x \mid x\%2 = 0,\ x \in D\}$$

where % represents the modulo operation.

Functions get^q_P and put^q_P define the mapping for view-updating of the state-based approach; get^q_P produces the view $D = get^q_P(D_P)$, and put^q_P replects the update D' of D onto the soource as $D'_P = put^q_P(D_P, D)$. We can see that this bidirectional transformation $\langle get^q_P, put^q_P\rangle$ satisfies the round-tripping property and so is well-behaved.

Similarly, defined in Q:

$$get^p_Q(D_Q) = \{q \mid q\%3 = 0,\ q \in D_Q\}$$
$$put^p_Q(D_Q, D) = D_P \setminus get^p_Q(D_Q) \cup \{x \mid x\%3 = 0,\ x \in D\}.$$

Then, we use these for data sharing in a way that the intermediate data D represents shared data consisting elements in both of $get^q_P(D_P)$ and $get^p_Q(D_Q)$, i.e., $D = get^q_P(D_P) \cup get^p_Q(D_Q)$. In brief, D contains sextuple numbers, i.e., numbers divisible by 6, common to D_P and D_Q.

We can confirm by the state-based semantics that local updates in P and Q are faithfully reflected in both D_P and D_Q through the Dejima D if this condition holds.

Now, we are going to our operation-based sharing.

Recall that get^q_P tells us that P is willing to share double numbers with Q, and that get^p_Q tells us that Q is willing to share triple numbers with P. However, the *put* functions tell us that P will accept only double numbers, and Q will accept only triple numbers from the common intermediate data D.

A short story is here:

1. Start from $D_P = \{1, 2, 3, 4\}$ and $D_Q = \{2, 3, 4, 9\}$.
2. Network connection fails. They are consistent, i.e., $D_P \sim D_Q$ since $D = \{\}$.
3. Concurrently, P does $\cup\{6\}$ and Q does $\setminus\{4\}$.
4. Connection restored, and synchronization processes begin in P and Q independently.

Then, what happens in synchronization processes?

These operations are in fact *effectful* in that $\cup\{6\}$ is applied to D_P which does not contain 6, and $\setminus\{4\}$ is applied to D_Q which does contain 4.

In Step 3, P' s local data becomes $D'_P = D_P \cup \{6\} = \{1, 2, 3, 4, 6\}$, and Q' s local data becomes $D'_Q = D_Q \setminus \{4\} = \{2, 3, 9\}$.

Synchronization proceeds as

– Since $\cup\{6\} {}^q_P\!\rightarrowtail \cup \{6\} \hookrightarrow^p_Q$, 6 is added to D'_Q to produce $D''_Q = D'_Q \cup \{6\} = \{2, 3, 6, 9\}$.
– On the other direction, since $\setminus\{4\}$ in Q cannot be passed to ${}^p_Q\!\rightarrowtail$ because get^p_Q rejects 4, this operation does not arrive at P.

Thus, these synchronization processes concurrently done in P and Q lead P's data and Q's data to the consistent state, i.e., $D'_P \sim D''_Q$ with $D = \{6\}$.

Another story is here: In Step 3 above, what happens if "Q does $\backslash\{6\}$ instead of "Q does $\backslash\{4\}$?

Note that these operations are not effectful because $\backslash\{6\}$ here is applied to D_Q which does not contain 6. During the period of network failure, P' s local data becomes $D'_P = D_P \cup \{6\} = \{1, 2, 3, 4, 6\}$ as before, and Q' s local data remains at it has been because $D'_Q = D_Q \setminus \{6\} = \{2, 3, 4, 9\}$.

Synchronization proceeds as follows after the network connection is restored.

– $\cup\{6\}^q_P\rightarrowtail \cup \{6\}\hookleftarrow^p_Q$ causes changes $D''_Q = D'_Q \cup \{6\} = \{2, 3, 6, 9\}$ as the previous case.
– And since $\backslash\{6\}^p_Q\rightarrowtail \setminus \{6\}\hookleftarrow^q_P \setminus \{6\}$, P may produces a new state $D''_P = D'_P \setminus \{6\} = \{1, 2, 3, 4\}$.

If the synchronization in P proceeds as above, P loses 6 which was added in Step 3, while it is added to Q' s local data by Q's synchronization. This breaks the consistency of D''_P and D'_Q.

From these examples, we observe that the effectful set operations in concurrent updates is essential for conflict-free synchronization. They effectively avoid insertion/deletion conflicts in synchronization. This is an extension of the scheme for data sharing described in [12]. Here, we used transformations *get* and *put* at the gateways of the peers.

References

1. Asano, Y., Hu, Z., Ishihara, Y., Onizuka, M., Takeichi, M., and Yoshikawa, M.: Data Integration Models and Architectures for Service Alliances, *Proceedings of the 4th Workshop on Software Foundations for Data Interoperability (SFDI2020), CCIS1281*, Springer, 2020, pp. 152–164.
2. Bohannon, A., Foster, J. N., Pierce, B. C., Pilkiewicz, A., and Schmitt, A.: Boomerang: Resourceful lenses for string data, *POPL*, 2008, pp. 407–419.
3. Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C., and Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem, *ACM Transactions on Programming Languages and Systems*, Vol. 29, No. 3(2007), pp. 17.
4. Hidaka, S., Hu, Z., Inaba, K., Kato, H., Matsuda, K., and Nakano, K.: Bidirectionalizing graph transformations, *ICFP*, 2010, pp. 205–216.
5. Hu, Z., Schürr, A., Stevens, P., and Terwilliger, J. F.: Dagstuhl Seminar on Bidirectional Transformations (BX), *SIGMOD Record*, Vol. 40, No. 1(2011), pp. 35–39.
6. Hu, Z., Mu, S.-C., and Takeichi, M.: A Programmable Editor for Developing Structured Documents based on Bidirectional Transformations, *Higher-Order and Symbolic Computation*, Vol. 21, No. 1-2(2008), pp. 89–118.
7. Ishihara, Y., Kato, H., Nakano, K., Onizuka, M., and Sasaki, Y.: Toward BX-based Architecture for Controlling and Sharing Distributed Data, *2019 IEEE International Conference on Big Data and Smart Computing (BigComp)*, 2019, pp. 1–5.
8. Ives, Z., Khandelwal, N., Kapur, A., and Cakir, M.: ORCHESTRA: Rapid, Collaborative Sharing of Dynamic Data, *CIDR*, 2005, pp. 107–118.

9. Karvounarakis, G., Green, T. J., Ives, Z. G., and l Tannen, V.: Collaborative data sharing via update exchange and provenance, *ACM Transactions on Database Systems*, Vol. 38, No. 3(2013), pp. 19:1–19:42.
10. Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M.: Conflict-free Replicated Data Types, *13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS 2011, Springer LNCS volume 6976, October 2011, pp. 386–400.
11. Takeichi, M.: BCDS Agent: An Architecture for Bidirectional Collaborative Data Sharing, *Computer Software, Japan Society for Software Science and Technology, Vol.38, No.3.*, 2021, pp. 41–57. Also available at `https://www.jstage.jst.go.jp/article/jssst/38/3/38_3_41/_pdf/-char/ja`.
12. Takeichi, M.: Conflict-free Collaborative Set Sharing for Distributed Systems, 2021. Technical Report available at `http://takeichimasato.net/blog/wp-content/uploads/2021/11/TR-CCSS.pdf`.

Appendix

Example of Homomorphic States Let $\langle \mathcal{A}, \circledcirc_A \rangle$ show the state of the door at the entrance and $\langle \mathcal{B}, \circledcirc_B \rangle$ show the state of the electric light of the entrance hall.

A={{DoorOpen}, {DoorClosed}}
$\circledcirc_A$: $\ominus$ for Open, $\otimes$ for Close, $\circledast$ for RingBell.

B={{LightLit}, {LightDim}}
$\circledcirc_B$: $\oplus$ for On, $\ominus$ for Off.

See the transitions in Fig.4 Note that every operator $\circledcirc$ takes an operand after it to validate application, e.g., RingBell operation $\circledast$ is valid only if the current state is DoorClosed.

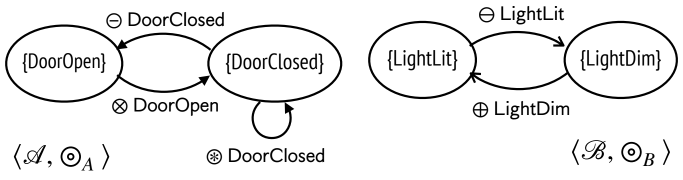


Fig. 4. Collaborative Working Door and Light

We can define the homomorphic mapping h from $\langle \mathcal{A}, \circledcirc_A \rangle$ to $\langle \mathcal{B}, \circledcirc_B \rangle$.

– For $\mathcal{A}$ to $\mathcal{B}$,

$$h(\{\mathsf{DoorOpen}\}) = \{\mathsf{LightLit}\}, h(\{\mathsf{DoorClosed}\}) = \{\mathsf{LightDim}\}$$

– For $\circledcirc_A$ to $\circledcirc_B$,

$$h(\ominus\{\mathsf{DoorClosed}\}) = \ominus\{\mathsf{LightLit}\}, h(\otimes\{\mathsf{DoorOpen}\}) = \oplus\{\mathsf{LightDim}\},$$
$$h(\circledast\{\mathsf{DoorClosed}\}) = !$$

Thus, the door and the light work together. Along with the homomorphism for the reverse direction gives us the collaborative updates of the states of the door and the light.