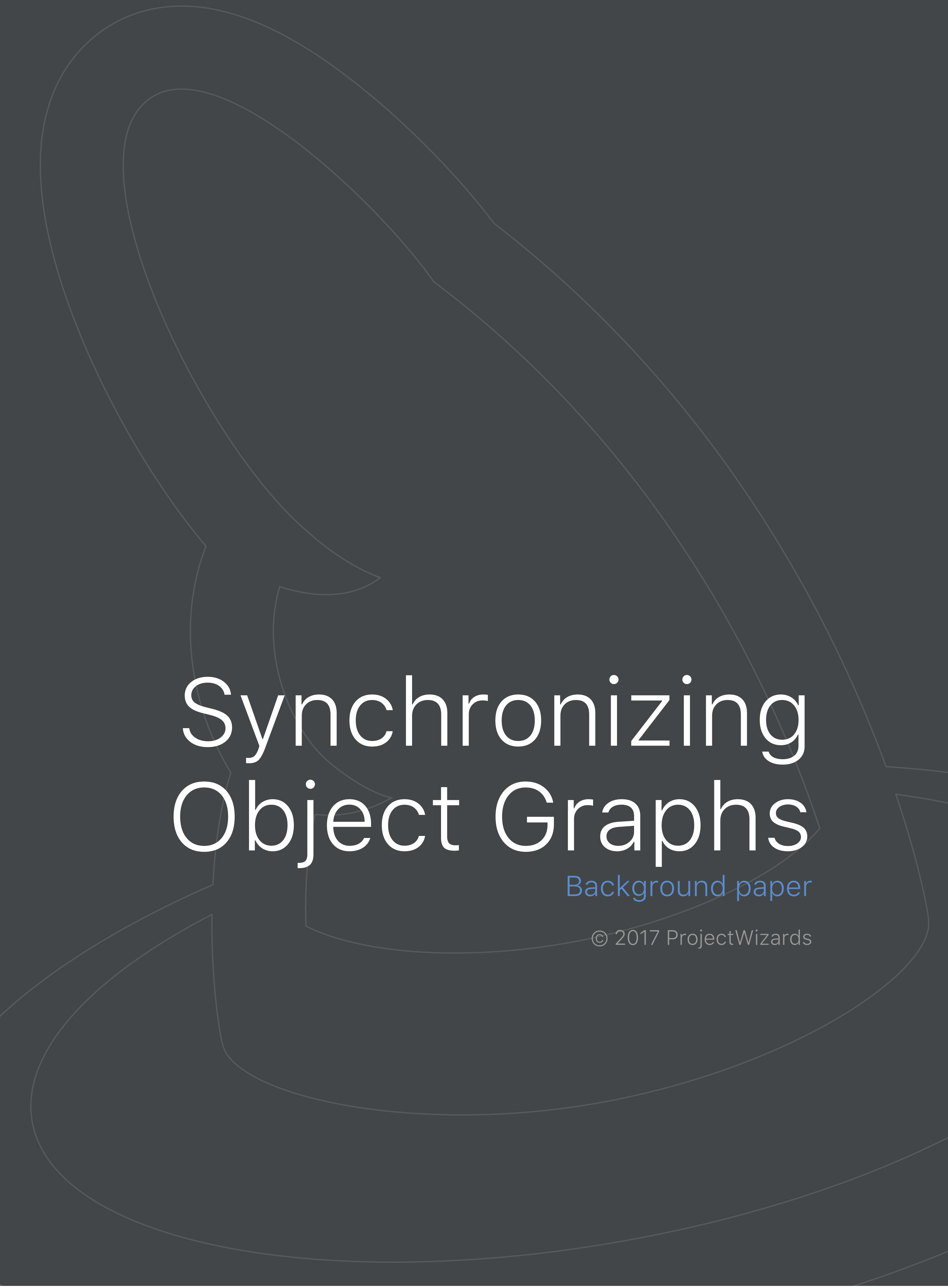

Synchronizing Object Graphs

Background paper

Synchronizing Object Graphs

Introduction to the Problem Space

Most modern businesses rely on computer applications for efficiently performing business functions. Common examples are:

- Accounting Software
- Customer Relationship Management Systems
- Project Management Software
- Resource Planning Software

(For more information see http://en.wikipedia.org/wiki/Enterprise_application_software)

These systems typically allow users to collaboratively work on shared business data. To make such a system easy to operate for its users, it is desirable that it offers the following features:

A) Unblocked Collaboration

A user working with the system should not block others from accessing or changing data. This should ideally include working on the same piece of information.

B) Consistent Merge And Intention Preservation

If multiple users are allowed to concurrently work on the same pieces of information, the system needs to be able to merge these changes. Merging needs to produce the same result for all users (convergence) and should strive to conserve as much of the users' original intentions as possible.

The merged data needs to be valid in the context of the application logic.

C) Offline Collaboration

Users want to access enterprise applications from their mobile devices like laptop computers, smartphones or tablets. These devices often operate in slow or unstable networking environments like cellular networks or are often completely offline for short periods like in subway trains or for longer periods like in airplanes or holiday homes. Users still expect to be able to work with their business applications even in these situations, therefore it needs to support offline collaboration.

If an application does not fulfill the unblocked collaboration requirement (A), blocking becomes even worse when users go offline for longer periods of time.

D) Undo Changes

In a business application, changes performed by one user can immediately affect others. As business applications tend to be complex, it is hard for users to always correctly predict the effect of their changes. It therefore greatly eases the use of an application, if changes can always be undone.

Traditional undo mechanisms are linear, meaning that changes have to be undone in the opposite order in which they were originally performed. For a user it is however desirable to be able to arbitrarily undo older changes without also having to undo the following actions. In a collaborative environment this so-called non-linear undo mechanism is mandatory, as changes from a user get interleaved with changes from others so that undoing only ones own changes requires skipping the changes of others.

E) Auto Save

All the changes a user performs on business data should be saved automatically without any interaction needed from the user. This is especially important for memory constrained mobile devices in which the work is likely to be interrupted. An incoming call for example should not result in data loss when the smartphone needs to terminate the business application.

Auto-saving should also enable the user to switch the device while he is working on a change: A user might want to begin his work in the office on a desktop computer and finish it on his way home on his smartphone.

F) Publishing independent from saving

When a user wants to perform a complex change to shared business data, he may want to perform multiple editing steps before he feels ready to share his changes with others. Therefore, publishing his changes to others ideally is independent from saving the changes and syncing them with other personal devices.

G) Audit Trail

If data is business critical, it is desirable to have a chronological record of the sequence of activities which changed the data. Therefore a business application should record all user actions as events and changes as a so called „audit trail". This increases the security of an application and even enables controlling workflows in which changes need to be reviewed.

Existing Solutions And Their Problems

There are many existing solutions for syncing business data between multiple users but we do not know of any solution which fulfills all of the above requirements. In our invention we focus on applications which are implemented using object-oriented programming languages and whose data is stored as an object-graph.

(For more information see: http://en.wikipedia.org/wiki/Object-oriented_programming or http://en.wikipedia.org/wiki/Object_graph)

File Base Syncing

Popular syncing services like DropBox or iCloud offer to sync changes which users perform on files. But as the semantics of the file contents are unknown to the syncing machinery, these services are not able to merge changes which are concurrently performed on a single file and leave it up to the application logic to solve the hard problem.

These systems are only a direct solution, if the business data can be separated into independent files, which is rarely the case. The file based syncing then only fulfills requirements A and C.

Central Online Server With Pessimistic Locking

Traditional enterprise applications are built around a central database server to which all users (clients) have a persistent online connection. The data stored on the server is divided into independent records which can be edited independently by different users. As long as a record is edited by a user, it gets locked, which means it cannot be edited by another user. Once the user has finished editing, the changes are immediately visible to other users and the record gets unlocked.

This common approach does not fulfill requirements A, B, C, D, E, F and G. It is also unsuitable for designs in which there are strong dependencies between objects because it needs to lock records independently.

http://en.wikipedia.org/wiki/Lock_(database)

Central Online Server With Optimistic Locking And Merging

A variant of the above approach is „optimistic locking" in which users are allowed to simultaneously edit the same pieces of data and the system is able to detect it. The system then needs to merge the individual changes, it however has no information for performing a merge which fulfills requirement B. Most commonly, merges are performed on a field-by-field basis.

The optimistic locking approach is common in object-relational mappers which persist data from object-graphs into relational databases.

It fulfills requirement A but not B, C, D, E, F and G

http://en.wikipedia.org/wiki/Optimistic_concurrency_control http://en.wikipedia.org/wiki/Object-relational_mapping

Key-Value Databases

In the last years, key-value or so-called NoSQL databases have become very popular and many web-based business applications are nowadays built around them. These databases also perform merging on a filed-by-field basis so that cross-dependencies between fields cannot be respected during a merge. This allows only for very simple data models. More sophisticated merging is possible with these databases but the problem is then shifted upwards into the application logic.

If fulfills requirements A and C but not B, D, E, F and G

The popular iCloud syncing service from Apple also offers a key-value database for syncing simple application data between iPhones and Macs.

http://en.wikipedia.org/wiki/NoSQL

https://developer.apple.com/library/mac/documentation/General/Conceptual/iCloudDesignGuide/Chapters/DesigningForKey-ValueDataIniCloud.html

System Prevalence

A successful approach for capturing more of the users' intentions is to store the whole history of user actions which led to the state of an object graph in addition to the object graph itself. This history is often called „log" or „journal".

When merging, action events originating from other users are applied to a user's object graph which improves the quality of merges. For example: If a two users concurrently add an object to a to-many relationship, a simple state-based merge would result in only one of the objects being added. A merge based on replaying of both events preserves the intentions and results in both objects being added.

However, in this scheme actions recorded by one user get transferred and applied to the state of another user which may not be the same as the one in which the action originated. This can lead to lots of consistency and convergence problems.

If prepared well, the log can enable linear undo by replaying the recorded actions in reverse. But it still does not allow for undoing changes in non-linear fashion. The log can also serve as the source for audit trail information.

The system prevalence pattern fulfills requirements A, C and G but not B, D, E, and F

Popular cloud syncing services like iCloud or DropBox offer stores which are implemented using the system prevalence pattern.

http://en.wikipedia.org/wiki/System_Prevalence http://en.wikipedia.org/wiki/Journaling_file_system https://www.dropbox.com/developers/datastore https://developer.apple.com/library/ios/documentation/General/Conceptual/iCloudDesignGuide/Chapters/DesignForCoreDataIniCloud.html

Differential Synchronization

Even without an explicit event log, it is possible to derive the user actions by performing a delta comparison of two object-graph states. Such derived actions may capture the user's original intention well if the differences are small. This immediately requires that clients are permanently connected to a server. Offline work would lead to greater differences from which the original actions cannot be derived reliably. A differential synchronization system also needs to be able to apply derived difference actions to a state which does not equal the original state from which the difference was computed. This is called „fuzzy patching". However, fuzzy patching is error prone and may result in different object graph states for different users.

http://en.wikipedia.org/wiki/Diff http://research.google.com/pubs/pub35605.html

Differential synchronization systems can fulfill requirement A and under some circumstances B, but not C, D, E,F and G

Operational Transformation

Operation Transformation (OT) is an extension of the System Prevalence pattern in which – in addition to maintaining the object graph – user actions get recorded into a log of operations. These operations can be transferred to other users, but are not directly applied to their object graph. They are first transformed against other operations to bring them into the correct context as the object graph of the receiving user might differ from the one in which the operations were originally recorded. (For a simple example using text documents see: http://en.wikipedia.org/wiki/Operational_transformation#Basics)

The challenge of implementing a working OT scheme lies in performing the following tasks:

Define a set of operations which can record all possible user actions and their undo inverse Define transformations between all possible operation pair combinations which guarantee convergence and preserve the user intention as much as possible. Define an integration algorithm which determines which transformations are to be applied to a given operation for bringing it into the desired operation context. The algorithm also needs to guarantee that custom validation requirements of the individual business logic do not get violated.

OT was originally developed for collaborative real-time plain-text editors but has since been applied to various other applications. But we currently do not know of any existing OT scheme which can successfully be applied to object-graphs of enterprise applications in the way described in the introduction to the problem space. For an overview of existing OT applications and academic papers see: http://en.wikipedia.org/wiki/Operational_transformation

OT solutions can fulfill requirements A, B, C, D and G, although the existing schemes make it very hard to implement operations which guarantee a correct undo behavior (requirement D).

For more information see: http://en.wikipedia.org/wiki/Operational_transformation http://www3.ntu.edu.sg/home/czsun/projects/otfaq/ http://googledrive.blogspot.de/2010/09/whats-different-about-new-google-docs.html

Our Solution

The following solution is the result of an unique development and filed for patent. It enables the online and offline editing of structured data with automatic synchronisation by means of a server or a cloud service (e. g. Dropbox)

- Title: "Invention for Synchronizing Object Graphs"
- Authors: Kai Brüning & Frank Illenberger for ProjectWizards GmbH
- Filed: in the USA on August 19, 2014 under application number 14/462,944.

Our invention fulfills the full set of requirements outlined above. It is based on the well-known concepts of Operational Transformation, Branching and System Prevalence.

Operational Transformation (OT)

Operational Transformation are a topic of active research (see part 1 of this paper). The basic concept is well understood, but consent exists that creating a set of operations and transformations which fulfill the necessary requirements is a hard problem. These requirements depend partially on the details of the integration algorithm. Requiring support for (non-linear) undo and redo does make the problem considerably harder.

Branching

Branches are widely used to store different states of a system in a retrievable and compact way. The prime example are source control management systems like Git or Subversion. To our knowledge we are the first to combine OT with branching.

System Prevalence Pattern

This pattern persists an object graph using a combination of object graph snapshots and a journal or log of user actions. Journaling file systems use a similar concept.

It is based on the concept of Operational Transformations (OT). OT are the topic of active research (see part 1 of this paper). The basic concept is well understood, but consent exists that creating a set of operations and transformations which fulfill the necessary requirements is a hard problem. These requirements depend partially on the details of the integration algorithm. Requiring support for (non-linear) undo and redo does make the problem considerably harder. We invented a set of operations suitable for object graphs together with transformations and an integration algorithm which support:

un-executing and re-executing arbitrary user actions anywhere in the history of operations. This forms the basis for implementing D (non-linear undo).

Our invention provides the full desired feature set outlined above. The system is based on Operational Transformations (OT), which provides for A, B (at least convergence) and C.

We use an implementation of Operational Transformations to support online and offline syncing, non-linear undo/redo and branching of an object graph. (and the Prevayler pattern for object graph persistence.)?

New Inventions

Our main invention is a set of operations and their transformations and an integration algorithm which support the above mentioned features. Additionally, the combination of branching with operational transformations and the use of the operations in the history to “shift” the object graph between branches is new.

Requirements for Operations and Integration Algorithm

Besides the basic OT requirement of ensuring convergence after syncing in all cases, we pose additional requirements:

- Preservation of basic object graph consistency constraints under syncing. The business logic typically places many constraints on the object graph, such as count restrictions of to-many relationships or certain correlations between attribute values. Almost any such constraint which spans more than a single attribute can be violated by syncing between object graphs which are individually valid before the syncing. Most such violations must be solved by the business logic, but our solution provides a few basic guarantees:
    - Object ownership is respected: if an object is deleted, any objects owned by it have to be deleted, too.
    - Object trees are kept cycle-free.
- Undo-ability of every event: after an arbitrary event has been undone, the object graph must be in the same state as if the event had never happened.

Object Addressing

Our operations are applied to the objects of the object graph, with each object being addressed by a unique object identifier. This is in stark contrast to the addressing in typical text-oriented OT, which uses an index into a single text string. Index-based addressing leads to the hard-to-solve false-tie problem (see http://www3.ntu.edu.sg/home/czsun/projects/otfaq/#_Toc321146194).

Most objects are identified by universally unique IDs (UUIDs). This guarantees that objects created concurrently have separate identities. Operations on them do not need to be transformed against each other.

In some cases though it is not tolerable that concurrent object creation leads to different object identities. One example are join objects: objects whose purpose it is to create a connection between two other objects (examples in Merlin are assignments and dependencies). If join objects between the same pair of objects are created concurrently, there must be only one join object after syncing the changes together.

This is solved by the use of so-called "**dependent IDs**": object identifiers which depend on one or several other object IDs. In the case of the join objects the ID of the join object is created by concatenating the IDs of the joined objects. Dependent IDs are not restricted to join objects, they can be used whenever object identity is best described in relation to other objects.

In some cases the dependent ID of an object can change while the object exists: a join object can be moved from one object pair to another one while any additional properties of the object are preserved. Our solution supports this with the help of an extra operation type for object ID changes.

Another kind of non-standard object addressing are objects uniquely identified by a name (for example resources in Merlin). Again, concurrently created objects with the same name must be considered as being the same object after syncing. Changing the name of existing objects is then handled as an ID change.

Operation Set

All operations contain the identifier of the object they are targeting, an old value and a new value. Operations with identical old and new value are no-ops. Such operations are never created by recording, but can be the result of a transformation. Possible value types depend on the kind of operation.

We use the following operations:

- **Object Operation**: create or delete the target object. Values are booleans reflecting the existence of the object. That is, a create-operation has oldValue == NO and newValue == YES, a delete-operation vice versa. Object operations carry a snapshot of the object state (in the form of a dictionary of key/value pairs of the properties of the object).
- **Identifier Operation**: change the identifier of the target object to newValue. The old value is always the target object ID.
- **Attribute Operation**: change the value of an attribute of the target object from oldValue to newValue. The attribute is identified by a key string.
- **Relationship Operation**: a sub class of the attribute operation for changing relationships. The values are the identifiers of the related object.
- **Container Operation**: a sub class of the relationship operation for the parent relationship in object trees. Values are the complete ancestor chains from the direct parent to the root of the tree, expressed as an array of object identifiers.

Many user actions result in more than one operation. The operations for one user action are grouped into an **event**. Events are the smallest undoable unit.

Multiple events form a **commit**, which is the smallest saved unit: a commit is closed each time a save action is triggered, either by the user or via auto save. Commits are identified by a UUID.

Commits are organized in **branches**, with one master branch which starts from an empty object graph, and any number of user branches, which diverge from the master branch at a split point. Only full commits are ever considered for syncing and merging.

Integration Algorithm

The integration algorithm in an OT system is responsible for transforming concurrent operations against each other in such a way that they can be integrated into a common history. This is non-trivial because operations can be transformed against each other only if they are defined on the same **operation context**. Transformations must therefore be done in a carefully planned order to conform to this restriction and still result in a list of operations building on each other.

In many OT systems the integration algorithm is driven by a so-called vector clock. In the simplest case this is a pair of indices attached to operations which are to be synced which effectively describe the context of the operation (TODO: reference). While simple and efficient, vector clocks do not support arbitrary merges between branches.

Our integration algorithm proceeds as follows to merge the changes from branch S (source) to branch T (target):

- Determine the most recent common point shared by both branches using the commit **history digests**.
- Build "merge node" objects for each commit after the common point in both branches. Merge nodes are identified by a node ID which needs to be unique within a single merge session (a simple index number will do), with the condition that merge nodes for the same commit get the same node ID. Note that an existing commit (identified by its UUID) can occur in both branches after the common point due to earlier merges in different orders. Besides its events (unpacked from the originating commit) and its node ID, a merge node contains its context, that is the set of IDs of merge nodes prior to it in the branch history.
- Merge nodes with the same context can be transformed against each other, creating two new merge nodes which have the ID of the other node in their context.
- Transforming two merge nodes with identical context against each other boils down to transforming two linear operation histories originating at a common context against each other. This is done with two nested loops over both histories which transform each operation from one side against each one from the other side and always keep the resulting operations for the next step.
- Build a two level lookup structure for merge nodes: first key is the node ID, second key the context set.
- Use a recursive algorithm to transform a node to a given context. If the requested merge node does not yet exist, the algorithm uses the lookup structure to find the merge node with the given ID and a context which is a subset of the given context with the fewest missing node IDs. Then it tries to recursively get any of the missing nodes in the correct context to transform against. As long as any transformation can be performed, the algorithm repeats until it either created the desired node or returns failure.

- Determine the target context, that is the set of the node IDs of the nodes created from branch T. Iterate over the merge nodes created from branch S (skipping any which are already present in the target branch) and transforms each of them into the target context using the recursive algorithm from above. After each step, add the ID of the transformed node to the target context. While the recursive node transformation algorithm can fail on deeper levels depending on ordering, it does always succeed at the top level due to the structure of branches and merge nodes.

The runtime complexity of merges scales at least with the product of the operation counts on both sides of the merge. The recursive merge node algorithm adds on top of this by transforming some nodes more than once. Two observations can help to minimize the runtime complexity of merging:

- The commit history digest supports efficient lookup of the most recent common point. Up to this point, both branches have an identical history, that is the same commits in the same order. OT integration is context based, that is the order of operations in the history is irrelevant. The actual depth of a merge can therefore be reduced further if the two branches contain the same commits in different order up to a point.
- Individual merge nodes can contain the events of more than one commit as long as these commits are not ordered differently in the two branches. If the branches are completely disjunct after the merge start point, this results in just one merge node per branch and a trivial case for the recursive algorithm.

In our case the integration algorithm is also responsible for undoing and redoing events in the history. Events are our unit for reversion, single operations in the history can’t be reverted.

To revert an arbitrary event back in the history, inverse operations are created for its operations in reversed order. This list of inverse operations is transformed against all operations in the history from this point on, and the operations in the history are transformed against the inverse operations. The resulting inverse operations are performed against the object graph and then discarded. The reverted event is marked as unexecuted and remains in the history at its original position. Redo is achieved analogically.

The advantage of this scheme compared to e.g. AnyUndo (TODO: reference) is that inverse operations are never transformed against other inverse operations. Doing the latter would be a hard and very complex problem. We are not aware of any operation/transformation sets which would work correctly with the integration algorithm of AnyUndo. Furthermore, we need the removal of the effect of an unexecuted event from the history for our conflict event scheme (see below).

The disadvantage is the lost immutability of the history: when an event is undone, the complete history beginning with this event is transformed and thereby permanently changed. This needs careful consideration when working with branches (see below).

To preserve the basic object graph constraints mentioned above, some operations simply can’t be merged while allowing both to change the graph. As an example, assume a tree structure with root R and leafs A and B. The first user moves A under B (that is, B becomes parent of A), the second user concurrently moves B under A (that is, A becomes parent of B). When naïvely combined, the parent of A would be set to B and the parent of B to A, forming a loop and disconnecting both from the root object. Simply removing one of the operations during transformation would solve this problem, but violate the undo requirement: if the event containing the other operation is undone, the removed operation would have to take effect again.

We solve this by the introduction of conflict events. These events do not contain operations, but describe the executed state of two or more events which creates a conflict (it is possible that a conflict arises only if a certain event is NOT executed). To resolve the conflict, the execution state of one of the conflicting events is changed. The conflict event records which event is in a so-called “forced” state. When a conflict is discovered during merging, a conflict event is created and inserted somewhere between the conflicting events. Each event carries a list of conflict events it takes part in, which is updated accordingly.

Although conflicts are produced by operations, conflict resolution is done at event level because events are the smallest undoable unit.

Adjustments in the Business Logic

- **Sorted Relationships**: the naive approach to a user-arrangeable relationship is adding a sorting index to each object and updating these indices when the user changes the order. This approach would be susceptible to the false-tie problem; besides being inefficient due to the potential large number of index update operations for a single insert. This is solved by making the order “index” a floating point value, which allows insertions and order changes without renumbering all objects in most cases. Renumbering is never done after an object is removed from the relationships, thus avoiding the false-tie problem. Collisions of identical order numbers after syncing are still possible, but these are real ties and are solved by a secondary sorting criterium (e.g. the object ID). If renumbering becomes necessary and happens concurrently, the result after syncing is not necessarily intention preserving. For typical uses of sorted relationships in business applications this is acceptable.

Glossary

Log

The combination of all commits and branches belonging to one object graph. The log describes the complete history of an object graph. The graph can be recreated from the log.

Object Graph

A set of objects which are connected by relationships and carry attributes. In the context of this paper, "object graph" refers to the set of objects which also carry the business logic and describe the user-visible state of the system.

Business Logic

Code implemented on the object graph. This code describes the allowed actions of the object graph and provides the user interface layer with an API to use and manipulate the model layer.

Operation

The smallest unit of the log, describing an atomic change to the object graph. Operations are recorded when the user edits the object graph.

Operation Context

The set of all previous operations in the history is called the context of an operation. Operations make sense in their context only, two operations can be transformed against each other only if they have the same context.

Event

The smallest undoable unit. An event contains one or more operations. An event is typically created by a single action of the user.

Execution State

An event can either be executed or unexecuted. Unexecuted events are marked as such but remain in the history. Their effect is removed from the history and object graph.

Event ID

Identifier for an event. Combines the UUID of the commit the event belongs to with the index of the event inside its commit.

Commit

The smallest mergeable unit, containing one or more events. A commit is created whenever a save action is triggered, either by the user or via auto save. Commits are identified by a UUID.

History Digest

A hash value (SHA1) held on each commit. It is calculated from the combination of the previous commits history digest and the UUID of this commit. Thus commits with identical history digest are guaranteed to have an identical history (same commits in same order).

History

List of operations (grouped in events and commits) which define the current state of the object graph. The history is further organized in branches.

Branch

The commit history is organized in one master branch, which starts from an empty object graph, and any number of user branches, which diverge from the master branch at a split point. Branches describe independent developments of the object graph. The object graph can be switched between branches by shifting. User branches can diverge from the master branch only, sub branching from user branches is not supported.

Shifting

The act of changing the state of the object graph from one commit to another commit. Typically used to change the object graph from its saved state to the head of a branch (user or master).

Conflict Event

A special kind of event, which does not contain operations but describes a conflict between other events. A conflict event contains the event IDs of the conflicting events together with the execution states which result in the conflict. If the conflict is resolved by changing the execution state of one of the conflicting events, the conflict event records which event is in forced state.

Forced State

If the execution state of an event is changed via a conflict event to resolve the conflict, it is considered to be in forced state. Changing its execution state again is only possible after another of the conflicting events has been forced.

Object Identifier

Operations address objects in the object graph by unique identifiers. Most objects use UUIDs as identifiers, but other identifiers like a unique name are possible (see Dependent ID). In some cases objects can change their identifier, which is recorded as an identifier operation.

Dependent ID

An object identifier which is built using one or several IDs of other objects. These identifiers better express the identity of objects which depend on other objects, like join objects.

Recording

The act of creating operations to record changes done by the user to the object graph.

Property

From the point of view of syncing an object in the object graph consists of properties and is addressed by its identifier. Properties are identified by key strings and can be attributes or relationships.

Attribute

A property with a “primitive” value. Primitive in this context means that the value does not relate to other objects in the graph and that it can be copied into an operation.

Relationship

A property which has one or more other objects in the graph as value. Relationships must always be bidirectional, that is the target object of a relationship must itself have a relationship pointing back to the source object.

Transformation

Operations are transformed against other operations to be able to integrate concurrently recorded operations into a single history. Transformation is possible between any two operations which are defined on the same operation context (that is the same state of the object graph). Afterwards the transformed operation has the other operation in its context, that is it is now defined on the object graph with the other operation applied.