CIMSync Protocol Specification

Thomas L. Rodeheffer
Microsoft Research, Silicon Valley

June 19, 2009

Abstract

Cimbiosys is a novel peer-to-peer replication platform that permits each device to define its own content-based filter criteria. Cimbiosys attains two properties not achieved by previous systems: (1) eventually, every device stores exactly those items whose latest version meets its arbitrary filter criteria, independent of any hierarchical namespace and (2) eventually, every device can summarize its metadata in a compact form, with size proportional to the number of devices rather than the number of items. The first property is a matter of correctness; the second a matter of efficiency.

This report describes and presents a specification of the Cimbiosys synchronization protocol, *CIMSync*. The specification is written in TLA+ and checked with the TLC model checker.

Contents

Chapter 1

Introduction

Cimbiosys [2, 3, 4] is a peer-to-peer replication platform for sharing a *collection* of data *items* among a number of devices. Each device stores a *replica* of the collection. Since a device may participate in several collections or even may store multiple replicas of the same collection, we shift our attention from the devices to the replicas. We consider replicas as the active agents in the collection.

A replica can be a *full replica*, which is interested in all items in the collection, or a *partial replica*, which is interested in only a subset of the items. The subset is defined by a per-replica content-based *filter*. A filter can be interpreted as a query over the items contained in the collection. A *star filter* matches all items regardless of content. A full replica can be considered as having a star filter. A replica may change its filter, thus changing the subset of items in which it is interested.

A replica may create new items in the collection and may create updated *versions* of existing items. An updated version *supersedes* any existing version or versions that it was derived from. Although the latest, unsuperseded version of each item is what is ultimately important to the collection, each replica performs updates independently of the other replicas and so the collection is only *weakly consistent*. Replicas *synchronize* with each other from time to time in order to pass information about new versions from one replica to another. Cimbiosys does not attempt to maintain any ordering between updates to different items in the collection.

The main contribution of Cimbiosys is in demonstrating how to permit content-based filtering among peer replicas while providing the following two important system properties.

- *Eventual filter consistency:* Over time, each replica receives all items that fall into its interest set (that is, all items that match its content-based filter) and discards all items that fall out of its interest set, so that eventually a partial replica stores precisely those items of the entire collection whose current version is of interest.

  Eventual consistency has long been demanded by applications and provided in replication systems, but it is more challenging to ensure in a system that permits peer-to-peer synchronization between content-based partial replicas. Not only

may a replica change its filter, thus changing the subset of items that are of interest to the replica, but also a replica may update an item, thus changing the set of replicas that find the item of interest. Eventual filter consistency is a matter of *correctness*.

- *Eventual knowledge singularity:* The metadata that replicas exchange during synchronization, which is used to determine which new versions one replica needs to learn about from its more-informed peer, eventually converges to a size that is roughly proportional to the number of replicas in the system rather than the number of stored items, even for partial replicas.

  Eventual knowledge singularity conveys the importance of compact metadata in making efficient use of bandwidth and system resources. In particular, this property allows Cimbiosys to utilize brief intervals of connectivity between peer replicas and also permits more frequent exchanges between regular synchronization partners, thereby reducing convergence delays. Eventual knowledge singularity is a matter of *efficiency*.

This report describes and presents a specification of the Cimbiosys synchronization protocol, CIMSync. The specification is written in TLA+ and checked with the TLC model checker [1]. The remainder of this report is organized as follows. Chapter 2 presents an overview of the synchronization protocol using a simplified example. Chapter 3 describes the system concepts of Cimbiosys and the synchronization protocol. Chapter 4 gives a brief tour through the specification and discusses the model checking results. A listing of the full specification appears in Appendix A and model configuration files in Appendix B.

Acknowledgement

The Cimbiosys synchronization protocol was developed over several years by Douglas Terry, Venugopalan Ramasubramanian, Thomas Rodeheffer, Ted Wobber, Dan Peek, and Meg Walraed-Sullivan, as part of the Community Information Management project at Microsoft Research. Originally, we thought it would be a straightforward matter to extend existing total replication schemes to support content-based partial replication. Many, many dicussions went into discovering and addressing the various problems that came up. The protocol or parts of the protocol were implemented several times in different ways by different people. The final result is truly a group effort.

The author distilled the protocol into this TLA+ specification so that an accessible and fairly complete representation could be published. Any defects in the specification are solely the fault of the author.

Chapter 2

Overview

CIMSync is a simple, two-step synchronization protocol that involves two replicas, a *source replica* and a *target replica*. First, the target sends a request to the source, informing the source of the target's current filter and state of knowledge about the collection. Second, the source sends a response to the target, updating the target's state of knowledge about the collection via new versions and other information. CIMSync brings the target replica up-to-date with respect to the source replica. To fully synchronize two replicas, it is sufficient to repeat the protocol in the reverse direction.

Figure 2.1 shows a simplified example. Each replica has *knowledge* of a set of versions, a *filter* that specifies the contents of items it is interested in, and a *store* containing current versions of items. Items are designated as i, j, k. Replicas are designated as A, B, C. Versions are designated as $A1$, $A2$, $C1$. Contents of items are abbreviated as simply w, x, y. A filter is specified by showing the set of possible item contents that it matches.

In the first step of the synchronization protocol, the target replica A sends its current knowledge and filter to replica B, requesting new information. Currently, replica A knows about versions $A1$, $A2$, $B1$, and $C1$. The replica's knowledge includes versions it stores, versions that it knows have been superseded, and versions that it knows do not match its filter.

In this example, replica A is storing three versions: $A1$, $A2$, and $C1$. Version $A1$ is a version of item i that has contents w; version $A2$ is a version of item j that has contents w, and version $C1$ is a version of item k that has contents x. Since replica A knows about version $B1$ but does not store it, version $B1$ must either be superseded or not match replica A's filter. Replica A's filter matches contents w and x. Note that it is not possible for replica A to tell what item $B1$ might be a version of. All replica A knows is that version $B1$ is either superseded or does not match A's filter.

In the second step of the synchronization protocol, the source replica B processes the request from A and examines its knowledge and stored versions in light of A's current knowledge and filter. Based on this examination, three types of information are sent back to replica A: new versions of items of interest to A, *move-outs* for items that A is storing but which have been updated and the new version is not of interest to A, and *learned knowledge*.

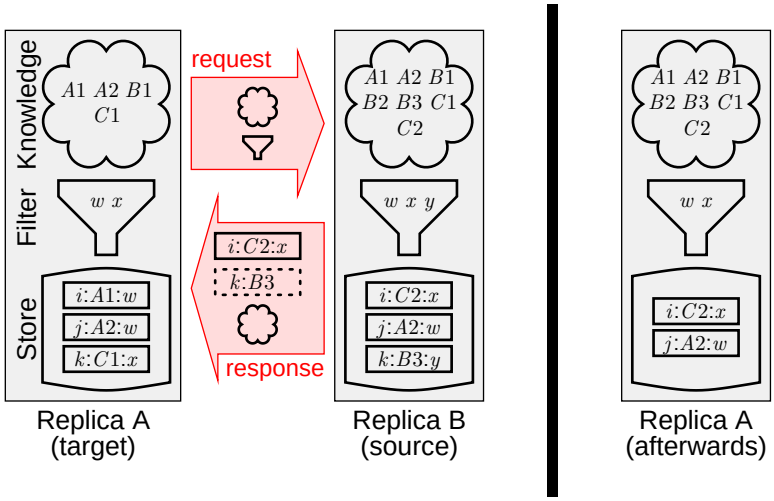

Figure 2.1: Cimbiosys synchronization protocol (simplified).

In this example, replica B identifies version $C2$ of item i as a new version of interest to replica A, determines that a move-out is needed for item k, and then informs replica A that it has learned everything that replica B knows.

Finally, the target replica A processes the response from the source and updates its knowledge and store accordingly. The updates from this processing are considered to be performed as a single atomic transaction. The resulting state of replica A is shown on the right-hand side of Figure 2.1. Observe that replica A now has the same knowledge as replica B and has the same set of versions in its store, except for item k, the latest version of which does not match replica A's filter.

Many essential details have been omitted from this simplified illustration. The next chapter goes into much more detail.

Chapter 3

System concepts

In this chapter we describe the system concepts of Cimbiosys, laying the groundwork for the synchronization protocol specification which appears in Appendix A. Since the specification explains all details of the protocol at a certain level of abstraction, the trick is to find a level of abstraction that provides insight and clarity without getting bogged down in endless detail.

The system concepts described here are structured according to the protocol specification in Appendix A. This structure differs somewhat from the published description of the Cimbiosys implementation [3], being more abstract in some ways and more detailed in others. The most significant change appears in Section 3.6, where we provide a different and more detailed description of a replica's store and knowledge. Related changes appear in our description of the synchronization protocol messages in Section 3.7. Replicas in Cimbiosys have content-based filters that can be changed at will, and this causes interesting problems regarding the perpetuation of updates, the perpetuation of conflicts, and the compaction of knowledge.

3.1 Collection

A *collection* is a set of data items that share a database schema. The collection is replicated over a number of *replicas*, each of which stores a subset of the items. We say that each replica *participates* in the collection.

Since each participating replica can independently create new items and update existing items, the collection is *weakly consistent*. Replicas synchronize with each other to spread information and thus eventually bring all replicas into a consistent state.

3.2 Identifiers and versions

3.2.1 Replica and replica identifier

A *replica* is the active entity that stores items in a collection. A replica can independently create new items and update existing items. Replicas synchronize with each other in a pair-wise fashion to spread information about items to all replicas participating in the collection.

A replica is labeled with a unique *replica identifier*. In this report we are not concerned with how replica identifiers are fashioned, with how replicas are created or

authenticated, or with how replicas locate or communicate with each other. We assume that these problems are solved, and that each replica is aware of the set of replica identifiers of all replicas in the collection. For simplicity, we use the letters A, B, C, and so on as replica identifiers.

3.2.2 Item and item identifier

An *item* is a particular data item in a collection. An item exists in *versions*: an item is created by creating an initial version and updated by creating updated versions.

An item is distinguished by a unique *item identifier* which labels all versions of the item. In this report, we are not concerned with how item identifiers are fashioned. For simplicity, we use the letters i, j, k, and so on as item identifiers.

3.2.3 Version

A *version* is a particular version of an item in a collection. A version is immutable once created. Formally, a version v is a tuple

$$\langle vi(v), ii(v), mw(v), cont(v) \rangle$$

where

- $vi(v)$ is a *version identifier* see Section 3.2.4), which uniquely labels the version,
- $ii(v)$ is an *item identifier* (see Section 3.2.2), which identifies which item this version is a version of,
- $mw(v)$ is a *made-with knowledge* (see Section 3.3.1), which indicates which versions of the same item this version supersedes, and
- $cont(v)$ is a *version content* (see Section 3.3.3), which is the data content of the version according to the database schema of the collection.

The version identifier and item identifier comprise an *extended identifier* (see Section 3.2.8). The version identifier, item identifier, and made-with knowledge comprise a *version header* (see Section 3.3.2).

3.2.4 Version identifier

Each version is labeled with a unique *version identifier*. The version identifier is fashioned by the replica that creates the version by concatenating its replica identifier and a per-replica *version number*, which increments with each version the replica creates. For replica A, we say that it creates versions $A1$, $A2$, $A3$, and so on. Replica B creates versions $B1$, $B2$, and so on. This design of version identifiers is part of Cimbiosys and is a common way that weakly-consistent replication systems provide for identifying versions.

3.2.5 Compact summary of knowledge

The reason for identifying versions by replica identifier and version number is that it makes it possible to have a compact summary of what versions a replica "knows" about. When a replica has somehow "examined" a set of versions and is storing all the unsuperseded versions that match its filter, we say that the replica "knows" about

that set of versions. One goal of the replication system is to spread knowledge until all replicas "know" about all versions that have ever been created. Quotation marks are used because this description omits many essential details.

Suppose that replica A has "examined" all versions created by replica B with version numbers in the range 1 through 198, all versions created by replica C with version numbers in the range 1 through 247, and all versions created by replica D with version numbers in the range 1 through 301. This is a lot of information, but replica A can summarize what it "knows" as simply

$$\{B1 \cdots 198, C1 \cdots 247, D1 \cdots 301\}$$

From this summary, a second replica can easily determine if it "knows" of any versions that are unknown to A. The concept of knowledge is central to Cimbiosys. Many more details will be described later.

3.2.6 Superseded versions

A version created as an update of one or more existing versions of an item is said to *supersede* those versions and, transitively, all versions that those versions supersede. As described in Section 3.3.1, made-with knowledge is used to determine which of two different versions of the same item supersedes the other.

In common with most weakly-consistent replication systems, in Cimbiosys versions that have been superseded are no longer of interest in the collection. The synchronization protocol spreads information about updates so that any superseded version is eventually removed from all replicas.

3.2.7 Conflict and resolution

Since replicas create versions independently, it is possible for two different versions of the same item to be created, neither of which supersedes the other. In such a case we say that the versions are in *conflict*.

We say that the conflict between two versions is *resolved* when yet another version is created that supersedes them both. Technically, the two conflicting versions are still in conflict, but since the collection retains only unsuperseded versions, the conflict is no longer of interest.

Typically, a weakly-consistent replication system has a *conflict resolution* process for discovering conflicts and resolving them. In this report, we are not concerned with what this process might be. We are only concerned that the supersession relation between versions be maintained properly so that any conflicts that might exist are perpetuated until they are resolved.

3.2.8 Extended identifier

Each version is a version of a particular item. We say that the version *pertains* to the item. Note, however, that it is not possible to tell from the version identifier what particular item the version pertains to.

Since we often have to associate a version identifier with the item identifier of the item the version pertains to, we form an *extended identifier* by concatenating the item identifier and the version identifier. The extended identifier designates a particular version just like a version identifier but it also designates the item the version pertains to.

3.3 Version metadata and content

Each version of an item consists of metadata, which is how Cimbiosys keeps track of the version, and content, which is what the user is interested in.

3.3.1 Made-with knowledge

Given two different versions of the same item, it is important to be able to determine which version, if either, supersedes the other. This is the purpose of *made-with knowledge*. If neither of two different versions of the same item supersedes the other, the versions are said to *conflict*.

As described in Section 3.2.3, each version v has a version identifier $vi(v)$ and made-with knowledge $mw(v)$. In Cimbiosys, made-with knowledge is a set of version identifiers. The made-with knowledge $mv(v)$ is defined as follows:

- Given version w such that $vi(w) \in mw(v)$, $w \neq v$, and w and v pertain to the same item, then v supersedes w.
- Version v supersedes no other versions.

Observe that the definition permits the made-with knowledge to include (1) version identifiers of versions that pertain to other items and (2) the version identifier of the superseding version itself. The reason for this freedom is that it makes it possible for replicas to compact the made-with knowledges of stored versions.

Made-with knowledge is a form of *knowledge* as defined in Section 3.4.

3.3.2 Version header

We call a version's metadata a *version header*. Although metadata may be compacted across the versions in a replica's store or the versions in a protocol message, in the abstract each version has its own metadata.

A version header for a version v consists of an extended identifier conjoined with made-with knowledge. The extended identifier contains the version identifier of v and the item identifier of the item to which v pertains. The made-with knowledge represents the set of versions superseded by v.

Version headers are important because in some places in the Cimbiosys synchronization protocol, only the metadata of a version needs to be conveyed and the actual content of the version is immaterial. This is the case for a direct move-out, as described in Section 3.7.3.

3.3.3 Version content

In addition to the version header metadata, a version contains *version content* which is the data in which the user is interested. The version content may be considered as a set of attribute-value pairs in some database schema. For simplicity in the specification, version content is represented by abstract values w, x, y and so on.

3.4 Forms of knowledge

Knowledge is a central concept in Cimbiosys and it shows up in many different places and in different forms. Unfortunately, this can be somewhat confusing. Even in this report, we use the term "knowledge" loosely within quotation marks because it is so convenient. However, careful use of terminology is essential to understanding the specification.

3.4.1 Knowledge

As used in the specification,

- *knowledge* is a set of version identifiers

What *knowledge* means is that you "know" something about the versions identfied by the version identifiers in the set. For example, in the case of made-with knowledge (see Section 3.3.1), you know that a given version supersedes every different version of the same item in the set.

When we say that a version identifier vi is in a knowledge k we mean simple set membership, $vi \in k$. Recall that an extended identifier, a version header, and a version all contain a version number vi as a component. By extension, when we say that an extended identifier, a version header, or a version is in a knowledge k, we mean that the component version identifier vi is in knowledge k.

3.4.2 Item-set knowledge

When a replica A is up-to-date with respect to all updates in the collection, having somehow "examined" all created versions and arranged to store the unsuperseded ones that match its filter, the replica can claim it "knows" about all the updates, as in the compact summary described in Section 3.2.5. However, there is a difficulty in conveying what A's "knowledge" means to another replica B. The difficulty arised because A "knows" versions with respect to A's filter, whereas B needs to "know" versions with respect to B's filter. And, in general, it is impossible to compute the relation between the two filters. Cimbiosys overcomes this difficulty through the use of *item-set knowledge*.

As used in the specification,

- *item-set knowledge* is a map from item identifier to knowledge

Recall that *knowledge* is a set of version identifiers. What *item-set knowledge* means is that for a given item identfier you "know" something about the versions identified by the version identifiers in the corresponding set.

The problem of conveying A's knowledge to B is solved by using item-set knowledge. Regardless of A's and B's filters, A can express what it "knows" as a map from item identifiers to what it "knows" about the specific versions of that item it is actually storing, and B can incorporate that knowledge, provided that B represents what it "knows" as item-set knowledge.

When we say that a version identifier vi is in an item identfier ii component of an item-set knowledge isk we mean simple set membership, $vi \in isk[ii]$. Recall that an extended identifier, a version header, and a version all contain a version number vi and an item identfier ii as components. By extension, when we say that an extended identifier, a version header, or a version is in an item-set knowledge isk, we mean that the component version identifier vi is in the component item identifier ii component of the item-set knowledge isk.

3.4.3 Star item-set knowledge

Recall that item-set knowledge is defined as a map from item identifiers to knowledge. An implementation would obviously attempt to encode item-set knowledge as

efficiently as possible, perhaps by grouping item identifiers that map to the same knowledge [3]. A particularly efficient grouping results when an item-set knowledge maps every item identifier to the same knowledge. Such an item-set knowledge we call *star item-set knowledge*.

3.4.4 Promotion of ordinary knowledge

Observe that ordinary knowledge, which is a set of version identifiers, is not parameterized by item identifier. Therefore ordinary knowledge can be promoted to star item-set knowledge via the obvious transformation.

3.5 Filters

3.5.1 Filter

Each replica has a content-based *filter* that selects the items of interest to that replica. A filter is a predicate on version content. The filter can be considered as a standing query over all the items in the collection. One goal of Cimbiosys is to execute this query so that the replica is storing the latest versions of items that match its filter.

In contrast to previous work in partial replication systems in which the interest set of a replica is based on a static labeling of items in a hierarchical namespace, a Cimbiosys filter is based on the current contents of items. Consequently, when an item is updated it can move from outside to inside the interest set of some replicas and from inside to outside the interest set of other replicas. When a replica changes its filter it can also cause some items to move from outside to inside the replica's interest set and other items to move from inside to outside the replica's interest set. Making sure that replicas are updated appropriately in these situations is one of the major problems addressed by the Cimbiosys synchronization protocol.

A replica's filter can be encoded in a query language and sent to another replica as part of the synchronization protocol. For example, in the synchronization request message, the target replica sends its filter to the source replica so that the source replica can limit its reply to contain only those versions that are of interest to the target. This saves much bandwidth when synchronizing with a target replica that has a narrow interest set.

For simplicity, in the specification we represent a filter as the set of version content it matches.

When we say a version v is in a filter f, we mean that the content of v matches the filter predicate.

3.5.2 Filter containment

When filter f_1 selects everything that filter f_2 selects, we say that f_1 *contains* f_2.

We expect that frequently a filter would be composed of a conjunction of query terms such as $rating > 2 \ and \ topic = music$. So in some cases it will be possible to compare filters via inspection.

Of course, in the general case, filter containment is undecidable. So, technically, what we have in Cimbiosys is *known containment*, when we know that filter f_1 contains filter f_2.

Having a filter containment relation, if we know it, helps in two situations. First, when a replica changes its filter, some items might move from outside to inside the

replica's interest set. As discussed in Section 3.5.4, if the old filter contains the new filter then knowledge can be retained that otherwise must be discarded.

Second, when a source replica is sending information to a target replica during synchronization, if the source replica's filter contains the target replica's filter, then the source has the advantage of being interested in everything the target is interested in. In such a case, the source can also inform the target about versions the source knows about but does not store. Such versions must not be of interest to the source. Since the source's filter contains the target's filter, the source can conclude that such versions would also not be of interest to the target.

3.5.3 Star filter

A *star filter* is a filter that is known to select everything. Therefore, a star filter is known to contain any other filter.

3.5.4 Filter change

In Cimbiosis, a replica can *change its filter*. This can result in some versions moving from outside to inside the replica's interest set and other versions moving from inside to outside the replica's interest set. There is a subtle difficulty with unsuperseded versions that move from outside to inside the replica's interest set.

In the following discussion, when we talk about what a replica "stores" and what it "knows", we are talking about the replica's *data store* and *data knowledge*. What is at risk when a replica changes its filter is its *indirect data knowledge*. See Sections 3.6.1 and 3.6.2.

Suppose that v is a version that moves from outside to inside a replica's interest set when the replica changes its filter. Now, before changing its filter, the replica might "know" about v, in that it knew that v had been created so that it was up-to-date in that regard, but the replica might not be storing it, because v was not of interest. The main reason why a replica would want to "know" about a version that it was not interested in storing is that by means of "knowing" such things it can obtain a more compact representation of knowledge, as discussed in Section 3.2.5.

However, after changing its filter, in the general case the replica could no longer claim to "know" about v. What if this version happened to be inside the new filter? Since the replica is not storing v, it cannot tell if this might be the case or not. Therefore the replica must discard its claim that it "knows" about v. Furthermore, the replica is in this position regarding any version of which it "knows" but does not store. The only safe course of action upon changing its filter is for the replica to discard its "knowledge" of all versions it does not store.

On the other hand, in the case that the old filter contains the new filter, there cannot be any versions that match the new filter without matching the old filter. So if the replica knows that the old filter contains the new filter, then it can retain everthing it "knows".

3.5.5 Filter shrink and unshrink

Changing a replica's filter when it is known that the old filter contains the new filter is called a *filter shrink*. As discussed in Section 3.5.4, in this case the replica can retain all of its base knowledge.

Changing a replica's filter when it is not known that the old filter contains the new filter is called a *filter unshrink*. This is the general case and is always a safe course of action. As discussed in Section 3.5.4, in this case the replica must discard "knowledge" of all versions it does not store.

3.5.6 Filter hierarchy

Since replicas typically only store versions that match their filter, and they can change their filters at any time, there is the problem of assuring that versions will eventually reach all replicas that are interested in them. Cimbiosys solves this problem by requiring that the replicas eventually form a *filter hierarchy*, a tree in which one replica with a star filter is chosen as the root and each other replica chooses a parent whose filter is known to include its own filter. Furthermore, each replica other than the root is required to synchronize regularly to and from its parent.

If the filter hierarchy remains stable long enough in the absence of updates, Cimbiosys will reach *filter consistency*, *knowledge singularity*, and *made-with singularity*.

- Filter consistency means that each replica's data store contains precisely the unsuperseded versions that match its filter, of all versions ever created.
- Knowledge singularity means that each replica's data knowledge is star item-set knowledge.
- Made-with singularity means that for each item with no unresolved conflicts, the made-with knowledge of all versions of that item at all replicas is identical.

Convergence to a stable filter hierarchy is provided by a policy layer on top of Cimbiosys. How it is achieved is not part of the specification. The specification only assumes that it happens eventually.

3.6 A replica's store and knowledge

Each replica has a store of versions and it "knows" something about how its store relates to updates performed by various replicas. Although this is a simple idea there are several complications.

- To perpetuate updates, a replica may be required temporarily to store a version that does not match its filter. For example, if a replica updates an existing version to a new version that does not match the replica's filter, the replica cannot immediately discard the new version because to do so would cause the update to be lost. A related case can result from a replica changing its filter.
- To compact knowledge via range encoding, Cimbiosys must be able eventually to account for each version number generated by every replica. This problem is related to perpetuating updates, although what must be perpetuated in this case is just the metadata.
- In the absence of conflicts, it is possible to simplify (densify) the most up-to-date version's made-with knowledge, which makes for a more efficient representation of metadata. However, with content-based partial replication, it is possible that of two conflicting versions of an item, one version matches a replica's filter and the other does not. This can lead to some bizarre situations.

The published description of the Cimbiosys implementation [3] deals with these complications only in part. The specification in this report deals with them in full.

We divide the replica's store into two parts:

- a *data store* and its associated *data knowledge*, which is concerned with keeping the most up-to-date versions of items that match the replica's filter, and
- an *auth store* and its associated *auth knowledge*, which is concerned with perpetuating updates.

In addition, a replica also maintains *conflict-free knowledge*, which is used in densifying made-with knowledge. These stores and knowledges are described next.

3.6.1 Data store and data knowledge

Each replica has a *data store* of versions and a corresponding *data knowledge* of item-set knowledge with the following properties.

- The replica "knows" everything it stores. Formally, for every version v in the data store, v and all of v's made-with knowledge is in the $ii(v)$ component of data knowledge. Recall that data knowledge is item-set knowledge, so it has a component for each item identifier.
- The replica stores every unsuperseded version matching its filter that it "knows". Formally, for every unsuperseded version v, if v matches the filter and v is in the data knowledge, then v must be in the data store.
- The replica "knows" no version that supersedes something it stores. Formally, for every created version v, if v is in the data knowledge but not in the data store, then there is no version w in the data store superseded by v.

The properties permit a replica's data store to contain versions that do not match its filter. The replica is permitted to discard such versions at any time. The idea is that the data store contains versions in the replica's interest set, plus perhaps a few extra that the replica has still lying around in its cache since the last filter change.

The properties permit a replica's data knowledge to contain versions in addition to the versions it stores. In particular, the replica's data knowledge may contain superseded versions and versions that do not match its filter. Versions that have been superseded and versions that do not match the replica's filter are not of interest to the replica, so of course the replica should not have to store them. However, it is important for the replica to know about such versions in order to demonstrate that it is fully up-to-date with respect to other replicas.

The data knowledge describes what the replica "knows" about the versions in its data store. The reason that a replica cannot have data knowledge of any version that supersedes something in its data store has to do with the case of a superseding version that does not match the replica's filter. The property is required for using *conflict-free knowledge* in *made-with knowledge densification* without regard for the replica's filter, as discussed in Section 3.6.7.

3.6.2 Direct and indirect data knowledge

A replica's data knowledge can be partitioned into *direct data knowledge* and *indirect data knowledge* as follows.

Direct data knowledge. This is data knowledge that can be derived by inspecting the version identifiers and made-with knowledge of the versions in the replica's data store. A replica always has direct data knowledge.

Indirect data knowledge. This is other data knowledge that a replica might have. Based on the properties of data knowledge, indirect data knowledge must be of versions that are either superseded or fail to match the replica's filter.

Indirect data knowledge is obtained in two ways. First, if a replica discards a version from its data store that does not match its filter, the direct data knowledge of that version that the replica used to have now becomes indirect data knowledge. Second, a replica can receive indirect data knowledge as *learned knowledge* as a result of synchronization.

As discussed in Section 3.5.4, a replica must discard all indirect data knowledge when it performs a *filter unshrink*.

3.6.3 Auth store and auth knowledge

Each replica has an *auth store* of versions and a corresponding *auth knowledge* of knowledge with the following properties.

- Permanence. For every unsuperseded version v, there is some replica that has v in its auth store.
- Compaction. For every created version v, there is some replica that has v in its auth knowledge.
- The replica "knows" everything it stores. Formally, for all versions v in the auth store, v is in auth knowledge.
- The replica stores every unsuperseded version it "knows". Formally, for every unsuperseded version v in auth knowledge, v is in the auth store.

The idea of the auth store is to hold every unsuperseded version somewhere, so that updates do not disappear from the collection even though they may currently only be known to replicas that are not interested in them. This is the reason for the permanence property.

The idea of the auth knowledge is to maintain knowledge of every created version somewhere, so that eventually knowledge can be compacted because all updates are known. The is the reason for the compaction property.

"Auth" stands for "authoritative". As originally conceived, the replica is authoritative for versions in its auth knowledge, any unsuperseded versions of which can be found in the replica's auth store. However, we have enough long words in the specification already, and "authoritative" is just one too many.

3.6.4 Auth concentration

In contrast to the data store and data knowledge, which the synchronization protocol tries to spread far and wide to all replicas, the synchronization protocol attempts to concentrate the auth store and auth knowledge. The idea is transfer a replica's auth store and knowledge to its parent in a *filter hierarchy*, with everything eventually ending up at the root. The filter hierarchy is described in Section 3.5.6.

Of course, the concentration is effective only when there is a filter hierarchy. How a filter hierarchy is achieved is not specified. The specification permits a replica to change its filter at any time. All that is assumed is that eventually a filter hierarchy is achieved and remains stable. Prior to that, the specification admits of relative chaos in terms of attempting to concentrate the auth store and knowledge. However, it is controlled chaos, in that nothing is lost.

3.6.5 Data knowledge compaction

Observe that auth knowledge is ordinary knowledge, which can be promoted to star item-set knowledge as described in Section 3.4.4. Hence auth knowledge is nicer than data knowledge, which in the general case is cumbersome item-set knowledge.

Careful study of the auth and data store and knowledge properties will show that it is permitted for a replica to "copy" its auth store and knowledge into its data store and knowledge, provided that the replica take care to discard superseded versions from its data store (which is always the case when adding to data knowledge).

The reason for performing such a "copy" is that it results in *data knowledge compaction*. The auth knowledge promotes into star item-set knowledge when copied into data knowledge, thus working towards *knowledge singularity*. This is especially effective if the auth store and knowledge have been concentrated at the root of the filter hierarchy, as discussed in Section 3.6.4.

3.6.6 Made-with knowledge densification

Recall that each version contains made-with knowledge that is used to determine the supersession relation between versions of the same item. Made-with knowledge is a set of version identifiers and in the general case nothing less than a set will do, since there are bizarre scenarios such as the following.

Suppose a replica learns about version v_1 of some item. It updates this version, creating version v_2 and spreads it to other replicas. Later, the replica changes its filter so that v_2 is no longer of interest, so it discards version v_2. Then it unshrinks its filter, so it has to forget all knowledge of v_2 and also of v_1. Subsequently, the replica learns again about the old version v_1 from some out-of-date replica. The replica now updates v_1 again, creating version v_3. Logically, v_2 and v_3 ought to be in conflict, since neither was made with knowledge of the other.

However, in most cases there will not be conflicts, and even when there are conflicts we would expect that eventually a conflict resolution process would render them uninteresting. It is cumbersome to have to maintain forever unique made-with knowledge on every version. In fact, when conflicts have been resolved for a given item it is legal to replace the made-with knowledge on the latest version of that item with the replica's data knowledge of that item. This replacement is called *made-with knowledge densification*.

Made-with knowledge densification is especially effective when the replica has achieved knowledge singularity and has the same data knowledge for every item.

3.6.7 Conflict-free knowledge

Given an item identifier ii, *conflict-free knowledge* of ii is a set of version identifiers $cfk(ii)$ with the following properties.

- If v_1 and v_2 are two conflicting versions of ii in $cfk(ii)$, then there exists a version v_0 of ii in $cfk(ii)$ such that v_0 supersedes both v_1 and v_2. In other words, there are no unresolved conflicts among versions of ii in $cfk(ii)$.
- A version v has been created corresponding to every version identifier in $cfk(ii)$. In other words, all the versions in $cfk(ii)$ have been created, so there will not ever be any unreolved conflicts among versions of ii in $cfk(ii)$.

Each replica maintains *conflict-free knowledge*, which is item-set knowledge mapping each item identifier ii to conflict-free knowledge of ii. Observe that it is safe to assume that conflict-free knowledge is empty. However, once some conflict-free knowledge is known, it is good forever, so over time it need never retreat.

Replicas learn conflict-free knowledge in two ways. First, it is sent via the synchronization protocol, so if a replica receives some that is better than what it currently has, it can (piecewise by item identifier) adopt the better conflict-free knowledge. Second, if a replica has a star filter, it can examine its data store. For items for which its data store contains no conflicts, the replica's corresponding data knowledge is conflict-free knowledge of that item.

Replicas use their conflict-free knowledge to perform *made-with knowledge densification*. (See Section 3.6.6.) Regardless of a replica's filter, if a replica's data store contains a version v such that

- $vi(v)$ is in the replica's conflict-free knowledge of $ii(v)$, and
- the replica's data knowledge of $ii(v)$ contains the replica's conflict-free knowledge of $ii(v)$

then it is permitted to replace the made-with knowledge of v with the replica's conflict-free knowledge of $ii(v)$. The first condition assures that v is in the conflict-free knowledge of $ii(v)$ and not superseded by anything the replica "knows". The second condition assures that the replica "knows" everything in the conflict-free knowledge of $ii(v)$. Note that the property that a replica's data knowledge cannot contain any version that supersedes a version in its data store gets used here. Consequently, v must be the only unsuperseded version of ii in the replica's conflict-free knowledge of $ii(v)$.

3.7 Synchronization protocol

The Cimbiosys synchronization protocol, *CIMSync*, is a simple, two-step protocol that involves two replicas, a *source replica* and a *target replica*.

First, the target sends a *synchronization request message* to the source, informing the source of the target's current filter and state of knowledge about the collection. Specifically, in addition to delivery information such as the message type and identification of the sender and receiver, the synchronization request message contains the following components:

- the target's filter,
- the target's filter unshrink number, a sequence number that increments whenever the target unshrinks its filter,
- the target's data knowledge, and
- an optional set of the extended identifiers of the versions in the target's data store.

Second, the source sends a *synchronization response message* to the target, updating the target's state of knowledge about the collection via new versions and other information. Specifically, in addition to delivery information such as the message type and identification of the sender and receiver, the synchronization request message contains the following components:

- a set of data versions,
- a set of transferred auth versions,
- transferred auth knowledge,
- accumulated conflict-free knowledge,
- the target's filter unshrink number,
- a set of direct move-outs,
- a set of indirect move-outs, and
- learned knowledge.

The protocol brings the target replica up-to-date with respect to the source replica. To fully synchronize two replicas, it is sufficient to repeat the protocol in the reverse direction.

We assume that the synchronization messages are delivered reliably. The issue of failure is beyond the scope of this report. Next we describe the concepts involved in the messages.

3.7.1 Data versions

Based on the target's filter and data knowledge, the source can determine which of the versions in its data store are (1) not known to the target and (2) of interest to the target. These versions the source sends back to the target as *data versions* to add to its data store.

3.7.2 Move-out

Because of partial replication, the target might have in its data store an obsolete version of an item whose updated version is not of interest to the target. If the source knows of the updated version, in order to bring the target up-to-date it must tell the target about the situation. We call this a *move-out*. The target has an obsolete version that it must move out of its data store in order to become up-to-date.

There are two cases:

- a *direct move-out*, in which the source has the updated version in its data store, and
- an *indirect move-out*, in which the source does not have the updated version in its data store.

3.7.3 Direct move-out

The *direct move-out* case is easier. The source has the updated version in its data store and it can inspect the version content to determine that indeed it does not match the target's filter. By checking the target's data knowledge, the source can determine that the target does not know about the updated version.

The source could speculate that there might be an obsolete version and that the target was storing it, but this seems like it would usually just be a waste of bandwidth. Instead, we require that the target also inform the source of the extended identifiers of the versions in its data store. With this information, the source can verify that the updated version actually does supersede a version that the target has in its data store. In this case, in order to bring the target up-to-date with regard to the updated version, the source must send a move-out to the target.

When sending a direct move-out, the source has the updated version in its data store so it can send the entire version header, containing the item identifier, version identifier, and made-with knowledge. The version content is of course not of interest to the target, since by the definition of a move-out, the version content does not match the target's filter. By sending the version header for a direct move-out, the target can determine what has been superseded and it can add the metadata to its indirect data knowledge.

3.7.4 Indirect move-out

The *indirect move-out* case is harder. Since the source does not have the updated version in its data store, it has access to neither the version content nor its metadata. One might wonder whether it was even worth worrying about this case, since the difficulties are large, but it turns out to be essential, as shown by the following example.

Suppose that there are three replicas, A, B, and C, arranged in a filter hierarchy (see Section 3.5.6) where C is the parent of B and B is the parent of A. By the definition of a filter hierarchy, C's filter contains B's filter and B's filter contains A's filter. Recall that eventual filter consistency is supposed to be assured provided that each replica other than the root regularly synchronizes to and from its parent.

Suppose that replica A creates an item that matches its filter and this item spreads via synchronization to replicas B and C. Next suppose that replica C updates the item so that it moves outside of B's filter. When B next synchronizes from C, C will send a direct move-out telling B to discard its obsolete version from its data store.

Note that replica A is now storing an obsolete version. Its parent replica B knows about the updated version but is does not have it in its data store, since it does not match B's filter. The question is how replica A ever becomes up-to-date. From the filter hierarchy rules, replica A must regularly synchronize to and from replica B, its parent, but it never need synchronize from replica C. The only way to resolve this situation is for replica B to send an indirect move-out to replica A.

There are several conditions required for an indirect move-out.

- The source's filter must contain the target's filter.
- There is an item identifier ii such that the source's data knowledge ii component contains the target's data knowledge ii component. In other words, the source knows everything about item ii that the target knows.
- The target has a version v of item ii in its data store. As for the direct move-out, we require that the target inform the source of the extended identifiers of the versions in its data store. With this information, the source can evaluate this condition.
- The source does not have version v in its data store.

If all these conditions are satisfied, the source can send an indirect move-out for version v to the target. Since the source does not store version v, it does not have access to the version header, but it does have the extended identifier for version v, which it can send.

3.7.5 The target's set of extended identifiers

In order to avoid speculative direct move-outs, and in order to enable indirect move-outs, we require that the target send the set of extended identifiers of the versions in its data store. However, even though this is required, optimizations are possible.

First, the target does not have to send the set every time. If the set is not sent, then the source will omit responding with move-outs and consequently also omit responding with learned knowledge.

Second, when requesting from a regular sychronization partner, the source could cache the set enabling the target to send only deltas and fingerprints most of the time.

3.7.6 Learned knowledge

In the ideal case, synchronization would cause the target to learn everything that the source knows. Specifically, the source's entire em data knowledge would be copied to the target. We call this *learned knowledge*.

Observe that the target gets a lot of data knowledge from the metadata of data versions and direct move-outs. This is not what we call learned knowledge here. Learned knowledge, if it can be sent, is the source's entire data knowledge.

The source must omit sending learned knowledge in the following cases.

- The source's filter does not contain the target's filter.
- The target omitted to send the set of extended identifiers for the versions in its data store.

3.7.7 Auth transfer

As described in Section 3.6.4, the synchronization protocol attempts to concentrate auth store and auth knowledge by transferring them up to parent replicas in a filter hierarchy. So whenever the target is the parent of the source, the source takes the entire contents of its auth store and auth knowledge and transfers it to the target.

When the target receives the response message, it incorporates the transferred auth store and auth knowledge into its auth store and auth knowledge.

3.7.8 Conflict-free knowledge accumulation

As described in Section 3.6.7, once some conflict-free knowledge is known, it is good forever. So the source always sends a copy of its current conflict-free knowledge.

When the target receives the response message, it performs a component-wise examination of the sent conflict-free knowledge to determine if it can improve any component of its current conflict-free knowledge.

3.7.9 Target filter skew

When the target receives the synchronization response message from the source, it can ingest the sent data versions, auth store, auth knowledge, and conflict-free knowledge without any hesitation. However, the sent move-outs and learned knowledge are based on the what the target's filter was at the time the target sent its request. The target could have changed its filter since then.

If the target has shrunk its filter, any move-outs and learned knowledge remain valid, since anything that was not of interest to the target is still not of interest. However, if the target has unshrunk its filter, this may not be true. Actually, the situation is even more complicated. If the target has unshrunk its filter at any time since sending the request, then there are scenarios in which the move-outs and learned knowledge might not be valid. We call this situation *target filter skew*.

In order to detect target filter skew, each replica maintains a counter that increments each time it unshrinks its filter. The value of this counter is sent in the request message and returned in the response message. If the value in the response message does not match the target's current value, them there is target filter skew and the target must ignore the move-outs and learned knowledge of the response.

Chapter 4

Tour of the specification

A listing of the TLA+ CIMSync protocol specification is given in Appendix A. The specification is divided into a large number of parts separated by horizontal rules. Basically, the specification defines options, data types, the initial state, a constraint, the next state relation, invariants, temporal assumptions, temporal properties, a state view, and finally the actual specification. Next we describe these parts in more detail.

Various options appear starting on page 25. Some options control how various constraints are interpreted in restricting the elaboration of the state graph. Other options introduce an intentional bug into the specification so that we can verify that the model checker actually finds a counterexample for that bug.

The data types follow fairly closely to the description of system concepts in Chapter 3. The main additions are in regard to the definition of a replica and of overall state. Much of the interesting detail of the specification appears in the definition of what happens with a replica. The overall state maintains extra information about the "truth" of the collection so that invariants can be checked with regard to "truth". The overall state also maintains some commentary about how the state was produced to aid in understanding a model execution history. This helps with debugging.

Invariants appear starting on page 51. Invariants are statements that are true in any reachable state.

Temporal assumptions appear starting on page 54 and temporal properties on page 55. Temporal properties are statements regarding an entire execution sequence that are true of any execution sequence that satisfies the temporal assumptions.

The state view appears on page 56 and the actual specification on page 57. The state view controls what the model checker considers when deciding when two states are the same state. We eliminate debugging information from the state view. As is typical when writing a TLA+ specification, the actual specification is a simple conjunction of the initial state, the next state relation, the liveness condition, and various temporal assumptions.

4.1 Model checking

Running the TLC model checker on a specification requires supplying a configuration file that provides model values and indicates which definitions form various parts of

configuration	sec	states	depth	error
B.4 BugAuthBounceForever	67	7421	24	Temporal properties were violated
B.5 BugContainFilter	44	21782	14	Invariant InvDataFilter is violated
B.6 BugLearnSend	466	172839	12	Invariant InvDataFilter is violated
B.7 BugLearnStore	297	116456	11	Invariant InvDataFilter is violated
B.8 BugOmitDiscardAuthSsin	8	736	20	Temporal properties were violated
B.9 BugOmitDiscardDataOof	39	5812	17	Temporal properties were violated
B.10 BugOmitIndMoveouts	1245	176737	12	Invariant InvHaveDataSuperseder is violated
B.11 BugOmitMoveouts	27	4754	16	Invariant InvHaveDataSuperseder is violated
B.12 BugOmitRebuildOnUnshrink	3	200	5	Invariant InvDataFilter is violated
B.13 BugUnionFreeisk	4	532	13	Invariant InvStoreMw is violated
B.14 BugUnshrinkLearn	15	9768	8	Invariant InvDataFilter is violated
B.15 BugUnshrinkMoveout	124	53999	18	Invariant InvDataFilter is violated

Table 4.1: Finding counterexamples for bugs.

the specification.

The interesting model values that have to be provided are the sets of item identifiers, replica identifiers, and version contents. These sets define how large a configuration is to be modeled.

Then there are a number of model values that constrain the state space in various ways by limiting the number of various actions that can be explored. Furthermore, there are separate per-replica and total system constraints. This is tedious, but this array of constraints is needed in order to guide the model checker into exploring paths that lead to counterexamples for various bugs.

Appendix B lists a number of configuration files. Section B.1, B.2, and B.3 list some small configurations that can be explored fully. No violations of invariants or of temporal properties were found for these small configurations.

4.2 Finding counterexamples for known bugs

When running the TLC model checker, it is nice to see that none of the invariants are violated for the size of the model that TLC can check. However, there is always the possibility that a bug lurks over the horizon. This is especially important for the CIM-Sync specification, since only very small configurations can be checked exhaustively. One way to get more confidence is to introduce a known bug on purpose and see if TLC finds a counterexample execution.

For this purpose, options were written into the specification to introduce various bugs. By setting the constraints to guide the model checker, a counterexample model execution can often be found fairly quickly. Table 4.1 gives the results. Each line in the table lists a configuration, the number of seconds of execution time required by the model checker to find a counterexample, the number of states examined and the depth reached, and the error reported by the model checker. The corresponding configuration files are listed in Appendix B.

Some of the bugs cause violations of temporal properties. For example, B.4 BugAuthBounceForever interferes with auth concentration (see Section 3.6.4). In this bug, the

auth store and auth knowledge are transferred whenever the target's filter contains the source's filter, rather than only when the target is the parent of the source. This is a bug, because if two non-root replicas have equal filters, their filters each contain the other and so auth store and knowledge could bounce forever between them, never reaching the root.

Bibliography

[1] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[2] P. Mahajan, R. Kotla, C. C. Marshall, V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, and T. Wobber. Effective and efficient compromise recovery for weakly consistent replication. In *EuroSys '09: Proceedings of the fourth ACM european conference on Computer systems*, pages 131–144, New York, NY, USA, 2009. ACM.

[3] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat. Cimbiosys: A platform for content-based partial replication. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*, pages 261–276, Apr. 2009.

[4] K. Veeraraghavan, V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, and T. Wobber. Fidelity-aware replication for mobile devices. In *Proceedings of MobiSys 2009*, June 2009.

Appendix A

CIMSync specification

MODULE *CIMSync*

Specification of the *CIM* Synchronization protocol.

EXTENDS *Naturals*, *Sequences*, *FiniteSets*, *TLC*

VARIABLE *state*

Useful definitions not part of standard TLA.

$a \supseteq b \triangleq b \subseteq a$ superset
$a \subset b \triangleq (a \subseteq b) \land (a \neq b)$ proper subset
$a \supset b \triangleq b \subset a$ proper superset

$Abort(x) \triangleq Assert(\text{FALSE}, x)$
$AbortType(x) \triangleq Abort(\langle \text{“type violation”}, x \rangle)$

OPTIONS

Options for this specification. These are defaults that can be overridden in a particular model. Some of these options control how various constraints restrict the elaboration of the state space graph. Other options introduce a bug so that we can verify that the model checker produces a counterexample.

Permit a replica to request a sync with itself. Since generally this does not produce any interesting interactions but does increase the state space, the default is to disable this option.

$OptSelfSync \triangleq \text{FALSE}$

What to count for a replica's parent change. Having each replica count its changes permits us to constrain the number of changes both per-replica and over the entire system. The default is 0, which means that we do not count changes.

$Parentinc \triangleq 0$ what to count for a parent change

Maximum per-replica values of various counters.

CONSTANT $MaxVersnum$ maximum per-replica version number
CONSTANT $MaxSyncact$ maximum per-replica number of active syncs
CONSTANT $MaxFilternum$ maximum per-replica filter changes
CONSTANT $MaxParentnum$ maximum per-replica parent changes

Maximum count of replicas that have non-zero values of various counters.

CONSTANT $MaxNzVersnum$ maximum replicas non-zero version number
CONSTANT $MaxNzSyncact$ maximum replicas non-zero active syncs
CONSTANT $MaxNzFilternum$ maximum replicas non-zero filter changes
CONSTANT $MaxNzParentnum$ maximum replicas non-zero parent changes

Maximum total values of various counters.

CONSTANT $MaxTotalVersnum$ maximum total version number
CONSTANT $MaxTotalSyncact$ maximum total number of active syncs
CONSTANT $MaxTotalFilternum$ maximum total filter changes
CONSTANT $MaxTotalParentnum$ maximum total parent changes

Omit to rebuild data item-set knowledge from stored versions when the replica unshrinks its filter. (Unshrink is a change that might not be a shrink.) This introduces a bug, because changing a filter in this way could make a version we know about come inside the new filter but the version is not stored because it is outside the old filter.

$BugOmitRebuildOnUnshrink \triangleq$ FALSE

Omit to perform moveouts. This introduces a bug, because moveouts are necessary in some situations.

$BugOmitMoveouts \triangleq$ FALSE

Assume that the target replica's filter does not unshrink between requesting a sync and processing the corresponding sync data. This introduces a bug, because if the target's filter does unshrink it could bring versions inside the target's filter that the source thought were outside, thus making the moveouts and consequently the learned item-set knowledge invalid.

$BugUnshrinkMoveout \triangleq$ FALSE
$BugUnshrinkIndMoveout \triangleq$ FALSE
$BugUnshrinkLearn \triangleq$ FALSE

Instead of checking that the target replica's filter did not unshrink between requesting a sync and processing the corresponding sync data, check that the target replica's filter at the former time contains the target replica's filter at the latter time. This introduces a bug, because the target's filter could shrink, causing the target to discard data versions that fell out of its filter, and then the target's filter could unshrink back to where it was.

$BugContainFilter \triangleq$ FALSE

Permit a source replica to send authoritative knowledge during sync whenever the target replica's filter contains the source replica's filter. This introduces a bug, because if both replicas' filters are equal they each contain the other, and the authority could bounce forever between both replicas without ever reaching the root.

$BugAuthBounceForever \triangleq$ FALSE

Omit to perform indirect moveouts. This introduces a bug, because indirect moveouts are necessary in some situations.

$BugOmitIndMoveouts \triangleq$ FALSE

During synchronization, for all data versions that the source stores (or sends), the target learns the source's knowledge of that item. These are bugs, because the source might know both versions involved in a conflict but only store one, whereas the other version matches the target's filter.

$BugLearnStore \triangleq$ FALSE

$BugLearnSend \triangleq$ FALSE

Omit to use authoritative knowledge to make star data item-set knowledge. This introduces a bug, because without using authoritative knowledge it is not in general possible to achieve eventual knowledge singularity.

$BugOmitMakeStar \triangleq$ FALSE

Omit to make conflict-free item-set knowledge. This introduces a bug, because without making conflict-free item-set knowledge it is not in general possible to achieve eventual madewith singularity.

$BugOmitMakeFreeisk \triangleq$ FALSE

Take the union of whatever conflict-free knowledge is sent with what I already have. This introduces a bug, because this is not correct.

$BugUnionFreeisk \triangleq$ FALSE

Omit to use conflict-free item-set knowledge to densify made-with knowledge. This introduces a bug, because without using conflict-free item-set knowledge it is not in general possible to achieve eventual made-with singularity.

$BugOmitDensifyMw \triangleq$ FALSE

Omit to discard out-of-filter versions from the data store. This introduces a bug, because it is not in general possible to achieve eventual filter consistency without discarding out-of-filter versions.

$BugOmitDiscardDataOof \triangleq$ FALSE

Omit to discard superseded versions from inside the auth store. This introduces a bug, because it is not in general possible to achieve eventual auth supersession without discarding such versions.

$BugOmitDiscardAuthSsin \triangleq$ FALSE

Basic types for this specification.

CONSTANT $Itemid$ item identifiers

CONSTANT $Content$ item contents

$Versnum \triangleq Nat$ version numbers
$Syncact \triangleq Nat$ outstanding sync requests
$Filternum \triangleq Nat$ filter change numbers
$Filterusn \triangleq Nat$ filter unshrink number
$Parentnum \triangleq Nat$ parent change numbers

REPLID

A replica id is modeled as an arbitrary value. We have some set of them. There is also $NullReplid$ which is an arbitrary value different from any replid.

CONSTANT $Replid$ set of replica ids

$NullReplid \triangleq$ CHOOSE $x : x \notin Replid$ something not a replica id
$ReplidOrNull \triangleq Replid \cup \{NullReplid\}$ replid or null

---

VERSID

A versid combines an author replid with a version number.

Each version of an item has a versid that is unique to that version. Therefore each versid pertains to a particular itemid. However, from the versid you cannot tell which itemid it is.

$Versid \triangleq [$
$ri : Replid,$ author of this versid
$vn : Versnum$ version number relative to the author
$]$

XTNDID

An $xtndid$ combines an $itemid$ with a $versid$.

Each version of an item has a $versid$ that is unique to that version. Therefore each $versid$ pertains to a particular $itemid$. However, from the $versid$ you cannot tell which $itemid$ it is. The $xtndid$ exhibits this relation.

$Xtndid \triangleq [$
$ii : Itemid,$
$vi : Versid$
$]$

KN

Kn is knowledge, a set of versids.

$Kn \triangleq$ SUBSET $Versid$

$EmptyKn \triangleq \{\}$

Compute the union of knowledges.

$UnionKns(ks) \triangleq$ UNION $\{k : k \in ks\}$

Determine if one knowledge contains another.

$GeqKn(k1,\ k2) \triangleq k1 \supseteq k2$

Better knowledge with other knowledge.

If the second knowledge is greater than the first, take it; otherwise stay with the first.

$BetterKn(k1,\ k2) \triangleq$ IF $GeqKn(k2,\ k1)$ THEN $k2$ ELSE $k1$

Make knowledge from a set of versids.

$KnMakeFromVersids(vis) \triangleq vis$

Make knowledge from a header or version.

$KnMakeFromHeader(h) \triangleq \{h.xi.vi\} \cup h.mw$
$KnMakeFromVersion(v) \triangleq KnMakeFromHeader(v.head)$

Determine if a versid, header, or version is in knowledge.

$IsVersidInKn(vi,\ k) \triangleq vi \in k$
$IsHeaderInKn(h,\ k) \triangleq IsVersidInKn(h.xi.vi,\ k)$
$IsVersionInKn(v,\ k) \triangleq IsHeaderInKn(v.head,\ k)$

HEADER

A $header$ combines an $xtndid$ with made-with knowledge.

Made-with knowledge is a set of versids. Each versid pertains to a particular itemid, but from the versid, you cannot tell which itemid it is.

There are several cases of versids that can appear in made-with knowledge.

(1) if the version xi is to supersede an earlier version vi of the same item, then vi must appear in the made-with knowledge.
(2) if vi is a version of some other item, then vi may appear in the made-with knowledge.
(3) the version of xi itself may appear in the made-with knowledge.

$Header \triangleq [$
$\quad xi \ : Xtndid,$
$\quad mw : Kn$
$]$

Determine if header ah supersedes header bh .

$$SupersedesHeader(ah,\ bh) \triangleq$$
$$\quad \wedge\ bh.xi.ii = ah.xi.ii \qquad \text{same itemid}$$
$$\quad \wedge\ bh.xi.vi \neq ah.xi.vi \qquad \text{different versid}$$
$$\quad \wedge\ IsVersidInKn(bh.xi.vi,\ ah.mw) \qquad ah \text{ made-with } bh$$

VERSION

A version combines a header with content.

$$Version \triangleq [$$
$$\quad head : Header,$$
$$\quad cont : Content$$
$$]$$

Determine if version av supersedes version bv . This is based entirely on their headers.

$$SupersedesVersion(av,\ bv) \triangleq SupersedesHeader(av.head,\ bv.head)$$

ISETKN

$Isetkn$ is item-set knowledge, a map from itemid to knowledge. Recall that knowledge is a set of versids.

Item-set knowledge isk means that for each itemid ii , we know all versids in $isk[ii]$. Recall that each versid pertains to a specific itemid, but from the versid you cannot tell which itemid it is.

There are two cases of versids that we can claim to know.

(1) if the versid vi pertains to itemid ii , then we can claim to know vi for itemid ii only if we actually know it, for whatever it means to know a versid.
(2) if the versid vi pertains to some other itemid than ii , then we can claim to know vi for itemid ii .

$$Isetkn \triangleq [Itemid \to Kn]$$

$$EmptyIsetkn \triangleq [ii \in Itemid \mapsto EmptyKn]$$

Compute the union of item-set knowledges.

$$UnionIsetkns(isks) \triangleq$$
$$\quad [ii \in Itemid \mapsto UnionKns(\{isk[ii] : isk \in isks\})]$$

Determine if one item-set knowledge contains another.

$$GeqIsetkn(aisk,\ bisk) \triangleq$$
$$\quad \forall\, ii \in Itemid : GeqKn(aisk[ii],\ bisk[ii])$$

Componentwise better the first item-set knowledge with the second.

$$BetterIsetkn(isk1,\ isk2) \triangleq$$

$[ii \in Itemid \mapsto BetterKn(isk1[ii],\ isk2[ii])]$

Make item-set knowledge from an itemid and knowledge, a header, version, set of headers, or set of versions.

$IsetknMakeFromItemidKn(ii0,\ k) \triangleq$
$[ii \in Itemid \mapsto \text{IF } ii = ii0 \text{ THEN } k \text{ ELSE } \{\}]$

$IsetknMakeFromHeader(h) \triangleq$
$IsetknMakeFromItemidKn(h.xi.ii,\ KnMakeFromHeader(h))$

$IsetknMakeFromVersion(v) \triangleq IsetknMakeFromHeader(v.head)$

$IsetknMakeFromHeaders(hs) \triangleq$
$UnionIsetkns(\{IsetknMakeFromHeader(h) : h \in hs\})$

$IsetknMakeFromVersions(vs) \triangleq$
$UnionIsetkns(\{IsetknMakeFromVersion(v) : v \in vs\})$

Make star item-set knowledge from knowledge.

$IsetknMakeStarFromKn(k) \triangleq [ii \in Itemid \mapsto k]$

Determine if we have star item-set knowledge.

$IsStarIsetkn(isk) \triangleq \forall\, ii1,\ ii2 \in Itemid : isk[ii1] = isk[ii2]$

Determine if a header or version is in item-set knowledge.

$IsHeaderInIsetkn(h,\ isk) \triangleq IsHeaderInKn(h,\ isk[h.xi.ii])$
$IsVersionInIsetkn(v,\ isk) \triangleq IsHeaderInIsetkn(v.head,\ isk)$

Get knowledge of an itemid from item-set knowledge.

$Isetkn\_GetItemidKn(isk,\ ii) \triangleq isk[ii]$

FILTER

We specify a filter by specifying the set of version content that the filter accepts.

An implementation would have to give a query predicate. Because comparison of predicates is not computable, an implementation cannot always tell whether one filter contains another. We do not specify that. However, with a rich enough set of version content, pairs of filters can be incomparable, which has the same effect as the comparison being incomputable.

$Filter \triangleq \text{SUBSET } Content$

A star filter.

$StarFilter \triangleq Content$

Determine if f is a star filter.

$IsStarFilter(f) \triangleq f = StarFilter$

Determine if a version is in a filter.

$IsVersionInFilter(v, f) \triangleq v.cont \in f$

Determine if one filter includes another, assuming we can tell. If we cannot tell, then the result is FALSE.

$GeqFilter(af, bf) \triangleq IsStarFilter(af) \lor bf \subseteq af$

Determine if one filter equals another, assuming we can tell. If we cannot tell, then the result is FALSE.

$EqFilter(af, bf) \triangleq GeqFilter(af, bf) \land GeqFilter(bf, af)$

MESSAGE

Message - request sync.

$MsgRequestSync \triangleq [$
- $type$: {"request sync"},
- $recvri$: $Replid$,
- $sendri$: $Replid$,
- tf : $Filter$, target's filter
- $tfusn$: $Filterusn$, target's filter unshrink number
- $tdataisk$: $Isetkn$, target's data item-set knowledge
- $txis$: SUBSET $Xtndid$, xtndids that target stores
- $txisyes$: BOOLEAN including target's full set of xtndids

$]$

Message $-$ $sync$ data.

$MsgSyncData \triangleq [$
- $type$: {"sync data"},
- $recvri$: $Replid$,
- $sendri$: $Replid$,
- $datavs$: SUBSET $Version$, data versions
- $authvs$: SUBSET $Version$, auth versions
- $authk$: Kn, transferred authoritative knowledge
- $freeisk$: $Isetkn$, accumulated conflict-free item-set knowledge
- tf : $Filter$, target's filter (only to exhibit a bug)
- $tfusn$: $Filterusn$, target's filter unshrink number
- $movehs$: SUBSET $Header$, direct move-outs
- $movevis$: SUBSET $Versid$, indirect move-outs
- $learn$: $Isetkn$ learned isetkn

$]$

Message

$Msg \triangleq \{\}$
$\cup\ MsgRequestSync$
$\cup\ MsgSyncData$

REPLICA

State of a replica.

The store consists of (1) a data versions store and (2) an auth versions store. It is legal, and in fact sometime required, for the same version to be in both the data versions store and the auth versions store. Presumably an implementation would optimize storage in such a case by using references.

We have knowledge about versions in our data versions store. If we know about a version that matches our filter, it must be in the data versions store. Nothing we have in our data versions store can be superseded by anything within our knowledge. It is legal for our data versions store to contain versions that do not match our filter; however, eventually such versions will be discarded.

We have authority about versions in our auth versions store. If we authority about a version that is not superseded, it must be in the auth versions store. It is okay to have authority about a superseded version. Authority and the corresponding versions are passed upward to our parent.

The data item-set knowledge comprises (1) direct data item-set and (2) indirect data item-set knowledge.

Direct data item-set knowledge comprises item-set knowledge about every version in the data store, whether or not that version matches our filter.

Indirect data item-set knowledge comprises whatever item-set knowledge we might have learned from move-outs or as declared learned item-set knowledge or from out-of-filter versions that we used to have in our data store. Indirect data item-set knowledge evaporates whenever we unshrink our filter.

To assist in model checking, we can limit the number of times a
replica is permitted to change its filter. For this purpose, each
replica maintains a counter *filternum* that is incremented on each filter change.

To assist in model checking, we can limit the number of times a
replica is permitted to change its parent. For this purpose, each
replica maintains a counter $parentnum$ that can be incremented on each parent change. The increment value is the model paramter $Parentinc$. If $Parentinc$ is 0, the number of parent changes is not tracked and therefore is unconstrained. Even so, the state graph does not grow without bound, since eventually all possible parents will be tried in all possible situations. If $Parentinc$ is 1, the number of parent changes is tracked and therefore also constrained.

Various replica operations construct new replica values with messages to be sent enqueued on $sendmsgq$. Then when the new value representing the replica is updated back into the main state, the sent messages are transferred to $recvmsgq$ in the proper destination
replica. This might seem complicated, but it makes writing the
replica operations much easier. Observe that $sendmsgq$ is always empty at the beginning of each action.

To assist in model checking, we limit the number of sync requests that a replica is permitted to launch without processing an answer. For this purpose, we track the number of outstanding sync requests.

$Replica \triangleq [$

ri	: $Replid$,	my replica id
$versnum$	: $Versnum$,	last version number used
$datavs$	: SUBSET $Version$,	data store

$dataisk$: $Isetkn$, data item-set knowledge

$authvs$: SUBSET $Version$, auth store
$authk$: Kn, auth knowledge

$freeisk$: $Isetkn$, conflict-free item-set knowledge

$filter$: $Filter$, current filter
$filterusn$: $Filterusn$, filter unshrink number
$filternum$: $Filternum$, number of filter changes

$parentri$: $ReplidOrNull$, my parent (might be null)
$parentnum$: $Parentnum$, number of parent changes

$recvmsgq$: $Seq(Msg)$, queue of messages received
$sendmsgq$: $Seq(Msg)$, queue of messages sent

$syncact$: $Syncact$ outstanding sync requests
]

Initial state for replica ri with parent pri and filter f .

$InitReplica(ri,\ pri,\ f) \triangleq$
[
$ri \mapsto ri$,
$versnum \mapsto 0$,

$datavs \mapsto \{\}$,
$dataisk \mapsto EmptyIsetkn$,

$authvs \mapsto \{\}$,
$authk \mapsto EmptyKn$,

$freeisk \mapsto EmptyIsetkn$,

$filter \mapsto f$,
$filterusn \mapsto 0$,
$filternum \mapsto 0$,

$parentri \mapsto pri$,
$parentnum \mapsto 0$,

$recvmsgq \mapsto \langle\rangle$,
$sendmsgq \mapsto \langle\rangle$,

$syncact \mapsto 0$
]

Determine if a replica thinks it is the root.

$DoesReplicaThinkItsRoot(r) \triangleq$
$\land\ r.parentri = NullReplid$ I do not have a parent
$\land\ IsStarFilter(r.filter)$ I have a star filter

Get the set of recv messages of a replica.

$GetRecvMsgsOfReplica(r) \triangleq \{r.recvmsgq[i] : i \in \text{DOMAIN } r.recvmsgq\}$

Get the set of all versions present in a replica.

$GetVersionsInReplica(r) \triangleq$
LET
$dmsgs \triangleq \{msg \in GetRecvMsgsOfReplica(r) : msg \in MsgSyncData\}$
$mvs \triangleq \text{UNION } \{msg.datavs \cup msg.authvs : msg \in dmsgs\}$
IN
$r.datavs \cup r.authvs \cup mvs$

Add new data headers to a replica.

Superseded versions are discarded from the data store. Headers are added to data item-set knowledge.

$ReplicaAddDataHeaders(r,\ hs) \triangleq$
LET
$keep(v) \triangleq \neg\exists\, h \in hs : SupersedesHeader(h,\ v.head)$
$isk \triangleq IsetknMakeFromHeaders(hs)$
IN
$[r \text{ EXCEPT}$
$!.datavs = \{v \in @ : keep(v)\},$
$!.dataisk = UnionIsetkns(\{@,\ isk\})$
$]$

Add versions to a replica's data store.

We add versions that are not already in the replica's data item-set knowledge. This ensures that we do not add a version for which we know a superseder. Then we add all the headers, which increases the replica's data item-set knowledge and discards superseded versions.

Note that there is no requirement to check that the versions match the replica's filter. Such versions can later be discarded at any time as out-of-filter versions.

$ReplicaAddDataVersions(r,\ vs) \triangleq$
LET
$addv(v) \triangleq \neg IsVersionInIsetkn(v,\ r.dataisk)$
IN
$[[r \mapsto r] \text{ EXCEPT}$
$!.r.datavs = @ \cup \{v \in vs : addv(v)\},$
$!.r = ReplicaAddDataHeaders(@,\ \{v.head : v \in vs\}),$
$!.r = @$
$].r$

Discard any versions from replica's data store that match a versid.

$$ReplicaDiscardDataVersids(r,\ vis) \triangleq$$
$$[r \text{ EXCEPT } !.datavs = \{v \in @ : v.head.xi.vi \notin vis\}]$$

Add versions to a replica's auth store.

$$ReplicaAddAuthVersions(r,\ vs,\ ak) \triangleq$$
$$[[r \mapsto r] \text{ EXCEPT}$$
$$!.r.authvs = @ \cup vs,$$
$$!.r.authk = @ \cup \{v.head.xi.vi : v \in vs\} \cup ak,$$
$$!.r = @$$
$$].r$$

Copy the auth store into the data store and copy the auth knowledge into star data item-set knowledge.

$$ReplicaMakeStar(r0) \triangleq$$
LET
$$copyauthdata(r) \triangleq ReplicaAddDataVersions(r,\ r.authvs)$$
$$copyauthknow(r) \triangleq [r \text{ EXCEPT } !.dataisk = UnionIsetkns(\{@,\ IsetknMakeStarFromKn(r.authk)\})]$$
IN
$$[[r \mapsto r0] \text{ EXCEPT}$$
$$!.r = copyauthdata(@),$$
$$!.r = copyauthknow(@),$$
$$!.r = @$$
$$].r$$

Discard out-of-filter versions from the data store.

$$ReplicaDiscardDataOof(r) \triangleq$$
$$[r \text{ EXCEPT } !.datavs = \{v \in @ : IsVersionInFilter(v,\ r.filter)\}]$$

Discard superseded versions from inside the auth store.

$$ReplicaDiscardAuthSsin(r) \triangleq$$
LET
$$ssin(v,\ vs1) \triangleq \exists\, v1 \in vs1 : SupersedesVersion(v1,\ v)$$
IN
$$[r \text{ EXCEPT } !.authvs = \{v \in @ : \neg ssin(v,\ @)\}]$$

Make conflict-free knowledge at a replica if possible.

$$ReplicaMakeFreeisk(r) \triangleq$$
LET

Versions of itemid ii in data store. Recall that the data store never contains any two versions such that one supersedes the other.

$dvsii(ii) \triangleq \{v \in r.datavs : v.head.xi.ii = ii\}$

Itemid ii is conflict-free in data store.

$freeii(ii) \triangleq \forall\, v1,\, v2 \in dvsii(ii) : v1 = v2$

Made conflict-free knowledge for itemid ii. Note that the fact that an itemid is conflict-free in our data store is meaningless unless we also have a star filter.

$mfreek(ii) \triangleq$ IF $\wedge\, IsStarFilter(r.filter)$
$\wedge\, freeii(ii)$
THEN $Isetkn\_GetItemidKn(r.dataisk,\, ii)$
ELSE $EmptyKn$

Made conflict-free item-set knowledge.

$mfreeisk \triangleq [ii \in Itemid \mapsto mfreek(ii)]$

IN

$[r$ EXCEPT $!.freeisk = BetterIsetkn(@,\, mfreeisk)]$

Use conflict-free knowledge to densify made-with knowledge.

Any version in the data store that is within conflict-free knowledge and about whose itemid we have data knowledge that contains the corresponding conflict-free knowledge can have its made-with knowledge replaced from the conflict-free knowledge.

Any version in the auth store that is also in the data store can be transformed in the same way.

$ReplicaDensifyMw(r) \triangleq$

LET

$datakv(v) \triangleq Isetkn\_GetItemidKn(r.dataisk,\, v.head.xi.ii)$
$freekv(v) \triangleq Isetkn\_GetItemidKn(r.freeisk,\, v.head.xi.ii)$

$datav(v) \triangleq$ IF $\wedge\, IsVersionInKn(v,\, freekv(v))$
$\wedge\, GeqKn(datakv(v),\, freekv(v))$
THEN $[v$ EXCEPT $!.head.mw = freekv(v)]$
ELSE v

$authv(v) \triangleq$ IF $\exists\, dv \in r.datavs : v.head.xi.vi = dv.head.xi.vi$
THEN $datav(v)$
ELSE v

IN

$[r$ EXCEPT
$!.datavs = \{datav(v) : v \in @\}$,
$!.authvs = \{authv(v) : v \in @\}$
$]$

Replica send message.

This is a simple operation. We write it up this way so that we get type checking on the message at a point near to where it is created, rather than much later.

$ReplicaSendMsg(r,\, msg) \triangleq$

IF $r \notin Replica$ THEN $AbortType(r)$ ELSE

IF $msg \notin Msg$ THEN $AbortType(msg)$ ELSE

$[r$ EXCEPT $!.sendmsgq = @ \circ \langle msg \rangle]$

Replica receive message - request sync.

$ReplicaRecvMsgRequestSync(r,\ msg) \triangleq$

LET

$tri \triangleq msg.sendri$
$tf \triangleq msg.tf$
$tfusn \triangleq msg.tfusn$
$tdisk \triangleq msg.tdataisk$
$txis \triangleq msg.txis$
$txisyes \triangleq msg.txisyes$

$tGeqF \triangleq GeqFilter(tf,\ r.filter)$
$tLeqF \triangleq GeqFilter(r.filter,\ tf)$

$vs \triangleq r.datavs$

Source and target data item-set knowledge.

$siik(ii) \triangleq Isetkn\_GetItemidKn(r.dataisk,\ ii)$
$tiik(ii) \triangleq Isetkn\_GetItemidKn(tdisk,\ ii)$

Determine if we should send authority on this sync reply.

$doa \triangleq$ IF $BugAuthBounceForever$ THEN
$\land\ tGeqF$
$\land \neg DoesReplicaThinkItsRoot(r)$
ELSE
$r.parentri = tri$

Compute auth versions and knowledge to send.

$authvs \triangleq$ IF doa THEN $r.authvs$ ELSE $\{\}$
$authk \triangleq$ IF doa THEN $r.authk$ ELSE $\{\}$

Compute data versions to send.

I should send every version in my data store that (a) matches the target filter and (b) is not already in the target data item-set knowledge.

$dosend(v) \triangleq \land\ IsVersionInFilter(v,\ tf)$
$\land \neg IsVersionInIsetkn(v,\ tdisk)$

$datavs \triangleq \{v \in vs : dosend(v)\}$

Compute direct move-out headers.

I should send a move-out derived from every version in my data store that (a) does not match the target filter and (b) supersedes something that the target has in its data store.

However, there is no need to send a move-out derived from a data version that I have already planned to send.

$$
\begin{array}{l}
movevs \triangleq \\
\quad \text{IF } BugOmitMoveouts \text{ THEN } \{\} \text{ ELSE} \\
\quad \{v \in vs : \\
\qquad \land \neg IsVersionInFilter(v,\ tf) \\
\qquad \land \exists\, xi \in txis : xi.vi \in v.head.mw \\
\quad \} \setminus datavs \\
movehs \triangleq \{v.head : v \in movevs\}
\end{array}
$$

Compute indirect move-out versids.

For every xtndid the target has in its data store, if
(a) my filter contains the target filter and
(b) I know everything about that itemid that the target knows and
(c) I do not store that xtndid and
(d) I am not already planning to send a superseder of that xtndid
then I should send an indirect move-out telling the target to discard that xtndid.

$$
\begin{array}{lll}
movexis \triangleq & & \\
\quad \text{IF } BugOmitMoveouts \text{ THEN } \{\} \text{ ELSE} & & \\
\quad \text{IF } BugOmitIndMoveouts \text{ THEN } \{\} \text{ ELSE} & & \\
\quad \{xi \in txis : & & \\
\qquad \land\ tLeqF & & \text{cover } t.filter \\
\qquad \land\ GeqKn(siik(xi.ii),\ tiik(xi.ii)) & & \text{know more of } ii \\
\qquad \land \neg\exists\, v \in vs : v.head.xi.vi = xi.vi & & \text{do not store version} \\
\qquad \land \neg\exists\, v \in datavs\ : xi.vi \in v.head.mw & & \text{no supersede version} \\
\qquad \land \neg\exists\, v \in movevs : xi.vi \in v.head.mw & & \text{no supersede move-out} \\
\quad \} & &
\end{array}
$$

$$
movevis \triangleq \{xi.vi : xi \in movexis\}
$$

Compute learned item-set knowledge.

$$
\begin{array}{l}
learn \triangleq \text{ LET} \\
\qquad\qquad iiv(v) \triangleq v.head.xi.ii \\
\qquad\qquad kv(v) \triangleq Isetkn\_GetItemidKn(r.dataisk,\ iiv(v)) \\
\qquad\qquad iskv(v) \triangleq IsetknMakeFromItemidKn(iiv(v),\ kv(v)) \\
\qquad \text{IN } \ UnionIsetkns \\
\quad (\{ \\
\qquad \text{If my filter contains the target filter and the target included his entire set of xtndids, then the target learns everything that I know.} \\
\qquad \text{IF } tLeqF \land txisyes \text{ THEN } r.dataisk \text{ ELSE } EmptyIsetkn,
\end{array}
$$

BUG: For all versions in my data versions store, the target learns my knowledge of that item. This is a bug, because I might know both versions involved in a conflict but only store one, whereas the other version matches the target's filter.

IF $BugLearnStore$ THEN $UnionIsetkns(\{iskv(v) : v \in r.datavs\})$
ELSE $EmptyIsetkn$,

BUG: For all versions I send, the target learns my knowledge of that item. This is a bug, because I might know both versions involved in a conflict but only store one, whereas the other version matches the target's filter.

IF $BugLearnSend$ THEN $UnionIsetkns(\{iskv(v) : v \in datavs\})$
ELSE $EmptyIsetkn$,

$EmptyIsetkn$
$\})$

Construct the message.

$dmsg \triangleq [$
$type \mapsto$ "sync data",
$recvri \mapsto tri$,
$sendri \mapsto r.ri$,
$datavs \mapsto datavs$,
$authvs \mapsto authvs$,
$authk \mapsto authk$,
$freeisk \mapsto r.freeisk$,
$tf \mapsto tf$,
$tfusn \mapsto tfusn$,
$movehs \mapsto movehs$,
$movevis \mapsto movevis$,
$learn \mapsto learn$
$]$

IN

Process.

$[[r \mapsto r]$ EXCEPT
$!.r = ReplicaSendMsg(@, dmsg)$, send the message
$!.r.authk = @ \setminus authk$, discard sent auth knowledge
$!.r.authvs = @ \setminus authvs$, discard sent auth versions
$!.r = @$
$].r$

Replica receive message $-$ *sync* data.

$ReplicaRecvMsgSyncData(r,\ msg0)\ \triangleq$

LET

$usn\ \triangleq$ IF $BugContainFilter$ THEN $GeqFilter(msg0.tf,\ r.filter)$ ELSE $r.filterusn = msg0.tfusn$

$msg\ \triangleq$

$[msg0$ EXCEPT

The source created this message based on an assumption of what my filter was. If at any time since then I ever unshrunk my filter, then what the source thinks I ought to move-out might now actually be inside my filter. And if I cannot accept the source's moveouts, then I cannot accept what the source tells me as learned item-set knowledge either.

Incidentally, it is not sufficient to check that my current filter is contained in my filter as assumed by the source. I could have shrunk my filter, discarded an out-of-filter version, and then unshrunk my filter back to what it was. In such a case, the source could send me learned knowledge that includes the discarded version but without sending me the discarded version.

$!.movehs\ =$ IF $usn \lor BugUnshrinkMoveout$ THEN $@$ ELSE $\{\},$
$!.movevis =$ IF $usn \lor BugUnshrinkIndMoveout$ THEN $@$ ELSE $\{\},$
$!.learn\ =$ IF $usn \lor BugUnshrinkLearn$ THEN $@$ ELSE $EmptyIsetkn$
$]$

IN

Process.

$[[r \mapsto r]$ EXCEPT
$!.r.syncact = @ - 1,$
$!.r = ReplicaAddDataVersions(@,\ msg.datavs),$
$!.r = ReplicaAddAuthVersions(@,\ msg.authvs,\ msg.authk),$
$!.r = ReplicaAddDataHeaders(@,\ msg.movehs),$
$!.r = ReplicaDiscardDataVersids(@,\ msg.movevis),$
$!.r.dataisk = UnionIsetkns(\{@,\ msg.learn\}),$

$!.r.freeisk\ =$ IF $BugUnionFreeisk$ THEN $UnionIsetkns(\{@,\ msg.freeisk\})$
ELSE $BetterIsetkn(@,\ msg.freeisk),$
$!.r = @$
$].r$

Replica receive message.

$ReplicaRecvMsg(r,\ msg)\ \triangleq$

CASE
$msg \in MsgRequestSync\ \rightarrow ReplicaRecvMsgRequestSync\ (r,\ msg)\ \square$
$msg \in MsgSyncData\ \rightarrow ReplicaRecvMsgSyncData\ (r,\ msg)\ \square$
OTHER $\rightarrow Assert($FALSE, $\langle$"unknown msg", $msg\rangle)$

STATE

$State \triangleq [$
  $replica : [Replid \to Replica],$ array of replicas
  $truth \ : \text{SUBSET } Version,$ all versions ever created
  $debug \ : Any$ helpful debugging stuff
$]$

The initial state with parent map $prim$ and filter map fm .

$InitState(prim,\ fm) \triangleq [$
  $replica \mapsto [ri \in Replid \mapsto InitReplica(ri,\ prim[ri],\ fm[ri])],$
  $truth \ \mapsto \{\},$
  $debug \ \mapsto \langle\rangle$
$]$

Get the replica for a given replid from a state.

$GetReplicaFromState(ri,\ s) \triangleq s.replica[ri]$

Count number of non-zeros given by replica id map.

$NzOfReplidMap(rm) \triangleq$
  LET $sum[ris \in \text{SUBSET } Replid,\ s \in Nat] \triangleq$
    LET
      $ri \triangleq \text{CHOOSE } ri \in ris : \text{TRUE}$
      $val(x) \triangleq \text{IF } x = 0 \text{ THEN } 0 \text{ ELSE } 1$
    IN
      IF $ris = \{\}$ THEN s ELSE $sum[ris \setminus \{ri\},\ s + val(rm[ri])]$
  IN
    $sum[Replid,\ 0]$

Count number of non-zero replicas in a state.

$NzFilternumInState(s) \triangleq$
  $NzOfReplidMap([ri \in Replid \mapsto s.replica[ri].filternum])$

$NzParentnumInState(s) \triangleq$
  $NzOfReplidMap([ri \in Replid \mapsto s.replica[ri].parentnum])$

$NzVersnumInState(s) \triangleq$
  $NzOfReplidMap([ri \in Replid \mapsto s.replica[ri].versnum])$

$NzSyncactInState(s) \triangleq$
  $NzOfReplidMap([ri \in Replid \mapsto s.replica[ri].syncact])$

Sum of nats given by replica id map.

$SumOfReplidMap(rm) \triangleq$
  LET $sum[ris \in \text{SUBSET } Replid,\ s \in Nat] \triangleq$
    LET $ri \triangleq \text{CHOOSE } ri \in ris : \text{TRUE}$
    IN IF $ris = \{\}$ THEN s ELSE $sum[ris \setminus \{ri\},\ s + rm[ri]]$
  IN $sum[Replid,\ 0]$

Total of various per-replica counters in a state.

$TotalFilternumInState(s) \triangleq$
  $SumOfReplidMap([ri \in Replid \mapsto s.replica[ri].filternum])$

$TotalParentnumInState(s) \triangleq$
  $SumOfReplidMap([ri \in Replid \mapsto s.replica[ri].parentnum])$

$TotalVersnumInState(s) \triangleq$
  $SumOfReplidMap([ri \in Replid \mapsto s.replica[ri].versnum])$

$TotalSyncactInState(s) \triangleq$
  $SumOfReplidMap([ri \in Replid \mapsto s.replica[ri].syncact])$

Verify that all replicas have formed a proper tree in state s. This encompasses the requirements:

(1) every child's filter is contained in its parent's filter,
(2) if a replica does not have a parent it has a star filter,
(3) every path of parent links is finite, and
(4) there is at most one replica without a parent.

$IsProperTreeInState(s) \triangleq$
  LET
    $r(ri) \triangleq GetReplicaFromState(ri,\ s)$ replica ri
    $rf(ri) \triangleq r(ri).filter$ ri's filter
    $pri(ri) \triangleq r(ri).parentri$ ri's parent replid
    $npri(ri) \triangleq pri(ri) = NullReplid$ ri has no parent
    $pr(ri) \triangleq r(pri(ri))$ parent replica
    $prf(ri) \triangleq pr(ri).filter$ parent's filter

    $FiniteParentPath[ri \in Replid,\ ris \in \text{SUBSET } Replid] \triangleq$
      IF $ri \in ris$ THEN FALSE ELSE
      IF $npri(ri)$ THEN TRUE ELSE
      $FiniteParentPath[pri(ri),\ ris \cup \{ri\}]$
  IN
  $\land \forall\, ri \in Replid : \neg npri(ri) \Rightarrow GeqFilter(prf(ri),\ rf(ri))$
  $\land \forall\, ri \in Replid : npri(ri) \Rightarrow IsStarFilter(rf(ri))$
  $\land \forall\, ri \in Replid : FiniteParentPath[ri,\ \{\}]$
  $\land \forall\, ri1,\ ri2 \in Replid : (npri(ri1) \land npri(ri2)) \Rightarrow ri1 = ri2$

Transfer all messages from replicas' sendmsg queues to the relevant recvmsg queues.

$$
\begin{array}{l}
StateTransferMsgs(s0) \triangleq \\
\quad \text{LET } xfer[s \in State] \triangleq \\
\quad\quad \text{LET} \\
\quad\quad\quad sq(ri) \triangleq s.replica[ri].sendmsgq \\
\quad\quad\quad sendris \triangleq \{ri \in Replid : Len(sq(ri)) > 0\} \\
\quad\quad\quad sendri \triangleq \text{CHOOSE } ri \in sendris : \text{TRUE} \\
\quad\quad\quad msg \triangleq Head(sq(sendri)) \\
\quad\quad\quad recvri \triangleq msg.recvri \\
\quad\quad\quad s1 \triangleq [s \text{ EXCEPT} \\
\quad\quad\quad\quad !.replica[sendri].sendmsgq = Tail(@), \\
\quad\quad\quad\quad !.replica[recvri].recvmsgq = @ \circ \langle msg \rangle \\
\quad\quad\quad\quad] \\
\quad\quad \text{IN} \\
\quad\quad\quad \text{IF } sendris = \{\} \text{ THEN } s \text{ ELSE } xfer[s1] \\
\quad \text{IN} \\
\quad\quad xfer[s0]
\end{array}
$$

Update a replica. Then transfer all messages.

$$StateUpdateReplica(s,\ r) \triangleq StateTransferMsgs([s \text{ EXCEPT } !.replica[r.ri] = r])$$

Add new version to truth.

$$StateAddVersionToTruth(s,\ v) \triangleq [s \text{ EXCEPT } !.truth = @ \cup \{v\}]$$

Add debug info.

$$StateAddDebug(s,\ x) \triangleq [s \text{ EXCEPT } !.debug = x]$$

CONSTRAIN

In order to restrict exploration to a reasonably small finite set of executions, we constrain various things that could lead to a large or unlimited number of distinct states. These things are

(1) the creation of a new version by a replica
(2) the number of unanswered syncs a replica has active
(3) the changing of a replica's filter
(4) the changing of a replica's parent

$$
\begin{array}{l}
Constrain(s) \triangleq \\
\quad \land\ NzFilternumInState(s) \leq MaxNzFilternum \\
\quad \land\ NzParentnumInState(s) \leq MaxNzParentnum \\
\quad \land\ NzSyncactInState(s) \leq MaxNzSyncact \\
\quad \land\ NzVersnumInState(s) \leq MaxNzVersnum
\end{array}
$$

$\land\ TotalFilternumInState(s) \leq MaxTotalFilternum$
$\land\ TotalParentnumInState(s) \leq MaxTotalParentnum$
$\land\ TotalSyncactInState(s) \leq MaxTotalSyncact$
$\land\ TotalVersnumInState(s) \leq MaxTotalVersnum$
$\land\ \forall\, ri \in Replid :$
LET
$r \triangleq GetReplicaFromState(ri,\ s)$
IN
$\land\ r.filternum \leq MaxFilternum$
$\land\ r.parentnum \leq MaxParentnum$
$\land\ r.syncact \leq MaxSyncact$
$\land\ r.versnum \leq MaxVersnum$

INITIAL STATE SPECIFICATION

Start with any assignment of parents and filters that makes a proper tree. Since replicas are all equivalent, it does not matter which one is the root, so we also restrict our consideration of initial states to those in which one particular (arbitrarily chosen) replica is the root.

$Init \triangleq$
$\exists\, prim \in [Replid \to ReplidOrNull] :$
$\exists\, fm \in [Replid \to Filter] :$
$\land\ state = InitState(prim,\ fm)$
$\land\ IsProperTreeInState(state)$
$\land$ LET
$rootri \triangleq$ CHOOSE $ri \in Replid$: TRUE — an arbitrary root
$rootr \triangleq GetReplicaFromState(rootri,\ state)$
IN
$rootr.parentri = NullReplid$

NEXT STATE RELATIONS

A replica changes its filter.

$NextChangeFilter(ri) \triangleq$
$\exists\, nf \in Filter :$
LET
$r \triangleq GetReplicaFromState(ri,\ state)$
$f \triangleq r.filter$
$shrink \triangleq GeqFilter(f,\ nf)$ — definitely shrinking the filter
$r1 \triangleq [r$ EXCEPT
$!.filter = nf,$
$!.filterusn = @ +$ IF $shrink$ THEN 0 ELSE $1,$

```
                !.filternum = @ + 1,
                !.dataisk =
                  IF shrink THEN @ ELSE
                  IF BugOmitRebuildOnUnshrink THEN @ ELSE
                  IsetknMakeFromVersions(r.datavs)
                ]
    debug ≜ IF shrink THEN "shrink filter" ELSE "unshrink filter"
    s1 ≜ StateUpdateReplica(state, r1)
    s2 ≜ StateAddDebug(s1, ⟨debug, ri, f, nf⟩)
  IN
  ∧ f ≠ nf                    suppress no change in filter
  ∧ state' = s2
```

A replica changes its parent.

```
NextChangeParent(ri) ≜
  ∃ npri ∈ ReplidOrNull :
  LET
    r   ≜ GetReplicaFromState(ri, state)
    pri ≜ r.parentri

    r1 ≜ [r EXCEPT
            !.parentri = npri,
            !.parentnum = @ + Parentinc
          ]

    s1 ≜ StateUpdateReplica(state, r1)
    s2 ≜ StateAddDebug(s1, ⟨"change parent", ri, pri, npri⟩)
  IN
  ∧ pri ≠ npri                suppress no change in parent
  ∧ pri ≠ ri                  do not pick self as parent
  ∧ state' = s2
```

Create a new item. We can use any content and any itemid.

Using an itemid that is already in use creates a root-level conflict, but is perfectly allowable in the specification. It has the benefit of allowing the model-checker to explore conflict scenarios without having first to establish the root version and then subsequently derive the conflicting versions.

```
NextCreateItem(ri) ≜
  ∃ cont ∈ Content :
  ∃ ii ∈ Itemid :
  LET
    r  ≜ GetReplicaFromState(ri, state)
    vn ≜ r.versnum + 1
    mw ≜ {}
```

$vi \triangleq [ri \mapsto r.ri,\ vn \mapsto vn\ ]$
$xi \triangleq [ii \mapsto ii,\ vi \mapsto vi\ ]$
$head \triangleq [xi \mapsto xi,\ mw \mapsto mw\ ]$
$v \triangleq [head \mapsto head,\ cont \mapsto cont]$

$r1 \triangleq [r \text{ EXCEPT } !.versnum = vn]$
$r2 \triangleq ReplicaAddAuthVersions(r1, \{v\}, EmptyKn)$

$s1 \triangleq StateUpdateReplica(state, r2)$
$s2 \triangleq StateAddVersionToTruth(s1, v)$
$s3 \triangleq StateAddDebug(s2, \langle \text{“create item”}, ri, v \rangle)$

IN

$\land\ state' = s3$

Update a version.

$NextUpdateVersion(ri) \triangleq$

LET

$r \triangleq GetReplicaFromState(ri, state)$
$vn \triangleq r.versnum + 1$

Itemids of versions in my data versions store.

$iis \triangleq \{v.head.xi.ii : v \in r.datavs\}$

IN

$\exists\, ii \in iis :$
$\exists\, uvs \in \text{SUBSET } \{v \in r.datavs : v.head.xi.ii = ii\} :$
$\exists\, cont \in Content :$

LET

Compute made-with knowledge of new version.

$mw \triangleq \text{UNION } \{\{v.head.xi.vi\} \cup v.head.mw : v \in uvs\}$

$vi \triangleq [ri \mapsto r.ri,\ vn \mapsto vn\ ]$
$xi \triangleq [ii \mapsto ii,\ vi \mapsto vi\ ]$
$head \triangleq [xi \mapsto xi,\ mw \mapsto mw\ ]$
$v \triangleq [head \mapsto head,\ cont \mapsto cont]$

$r1 \triangleq [r \text{ EXCEPT } !.versnum = vn]$
$r2 \triangleq ReplicaAddAuthVersions(r1, \{v\}, EmptyKn)$

$s1 \triangleq StateUpdateReplica(state, r2)$
$s2 \triangleq StateAddVersionToTruth(s1, v)$
$s3 \triangleq StateAddDebug(s2, \langle \text{“update version”}, ri, v \rangle)$

IN

$\land\ uvs \neq \{\}$ non-empty set of versions to update
$\land\ state' = s3$

Use authoritative knowledge to make star knowledge.

$NextMakeStar(ri) \triangleq$
LET
$r \triangleq GetReplicaFromState(ri,\ state)$
$r1 \triangleq ReplicaMakeStar(r)$
$s1 \triangleq StateUpdateReplica(state,\ r1)$
$s2 \triangleq StateAddDebug(s1,\ \langle$"make star"$,\ ri\rangle)$
IN
$\land \neg BugOmitMakeStar$
$\land r \neq r1$
$\land state' = s2$

Make conflict-free knowledge at a replica if possible.

$NextMakeFreeisk(ri) \triangleq$
LET
$r \triangleq GetReplicaFromState(ri,\ state)$
$r1 \triangleq ReplicaMakeFreeisk(r)$
$s1 \triangleq StateUpdateReplica(state,\ r1)$
$s2 \triangleq StateAddDebug(s1,\ \langle$"make freek"$,\ ri\rangle)$
IN
$\land \neg BugOmitMakeFreeisk$
$\land r \neq r1$
$\land state' = s2$

Use conflict-free knowledge to densify made-with knowledge.

$NextDensifyMw(ri) \triangleq$
LET
$r \triangleq GetReplicaFromState(ri,\ state)$
$r1 \triangleq ReplicaDensifyMw(r)$
$s1 \triangleq StateUpdateReplica(state,\ r1)$
$s2 \triangleq StateAddDebug(s1,\ \langle$"densify madewith"$,\ ri\rangle)$
IN
$\land \neg BugOmitDensifyMw$
$\land r \neq r1$
$\land state' = s2$

Discard out-of-filter versions from the data store.

$NextDiscardDataOof(ri) \triangleq$
LET
$r \triangleq GetReplicaFromState(ri,\ state)$
$r1 \triangleq ReplicaDiscardDataOof(r)$
$s1 \triangleq StateUpdateReplica(state,\ r1)$

```
    s2  ≜ StateAddDebug(s1, ⟨"discard data oof", ri⟩)
  IN
  ∧ ¬BugOmitDiscardDataOof
  ∧ r ≠ r1
  ∧ state' = s2
```

Discard superseded versions from inside the auth store.

```
NextDiscardAuthSsin(ri) ≜
  LET
    r   ≜ GetReplicaFromState(ri, state)
    r1  ≜ ReplicaDiscardAuthSsin(r)

    s1  ≜ StateUpdateReplica(state, r1)
    s2  ≜ StateAddDebug(s1, ⟨"discard auth ssin", ri⟩)
  IN
  ∧ ¬BugOmitDiscardAuthSsin
  ∧ r ≠ r1
  ∧ state' = s2
```

Request sync.

```
NextRequestSync(targetri, sourceri, txisyes) ≜
  LET
    r   ≜ GetReplicaFromState(targetri, state)
    r1  ≜ [r EXCEPT !.syncact = @ + 1]

    txis ≜ IF txisyes THEN {v.head.xi : v ∈ r.datavs} ELSE {}
    msg ≜ [
            type     ↦ "request sync",
            recvri   ↦ sourceri,
            sendri   ↦ r.ri,
            tf       ↦ r.filter,
            tfusn    ↦ r.filterusn,
            tdataisk ↦ r.dataisk,
            txis     ↦ txis,
            txisyes  ↦ txisyes
          ]
    r2  ≜ ReplicaSendMsg(r1, msg)

    s1  ≜ StateUpdateReplica(state, r2)
    s2  ≜ StateAddDebug(s1, ⟨"request sync", targetri, sourceri⟩)
  IN
  ∧ ¬OptSelfSync ⇒ targetri ≠ sourceri
  ∧ state' = s2
```

Initiate sync upto my parent. Provided I have a parent, he requests a sync from me. There is no need for my parent to send me his full set of xids.

$NextInitiateSyncUptoParent(ri) \triangleq$

LET

$r \triangleq GetReplicaFromState(ri, state)$

$pri \triangleq r.parentri$

IN

$\land\ pri \neq NullReplid$

$\land\ NextRequestSync(r.parentri, ri, \text{FALSE})$

Initiate sync from my parent. Provided I have a parent, I request a sync from him. I must send my full set of xis in order to get learned knowledge.

$NextInitiateSyncFromParent(ri) \triangleq$

LET

$r \triangleq GetReplicaFromState(ri, state)$

$pri \triangleq r.parentri$

IN

$\land\ pri \neq NullReplid$

$\land\ NextRequestSync(ri, r.parentri, \text{TRUE})$

Process message.

$NextProcessMsg(ri) \triangleq$

LET

$r \triangleq GetReplicaFromState(ri, state)$

$msg \triangleq Head(r.recvmsgq)$

$r1 \triangleq [r \text{ EXCEPT } !.recvmsgq = Tail(@)]$

$r2 \triangleq ReplicaRecvMsg(r1, msg)$

$s1 \triangleq StateUpdateReplica(state, r2)$

$s2 \triangleq StateAddDebug(s1, \langle \text{“process msg”}, ri, msg \rangle)$

IN

$\land\ Len(r.recvmsgq) > 0$

$\land\ state' = s2$

Put all the alternatives together into one next state relation.

Various replica bookkeeping next state relations have priority in order to hold down the state graph explosion during model checking.

$PrioMakeStar \triangleq \exists\, ri \in Replid : NextMakeStar(ri)$

$PrioMakeFreeisk \triangleq \exists\, ri \in Replid : NextMakeFreeisk(ri)$

$PrioDensifyMw \triangleq \exists\, ri \in Replid : NextDensifyMw(ri)$

$PrioDiscardDataOof \triangleq \exists\, ri \in Replid : NextDiscardDataOof(ri)$

$PrioDiscardAuthSsin \triangleq \exists\, ri \in Replid : NextDiscardAuthSsin(ri)$

$NextAny \triangleq$

IF ENABLED $(PrioMakeStar)$ THEN $PrioMakeStar$ ELSE

IF ENABLED $(PrioMakeFreeisk)$ THEN $PrioMakeFreeisk$ ELSE

IF ENABLED $(PrioDensifyMw)$ THEN $PrioDensifyMw$ ELSE
IF ENABLED $(PrioDiscardDataOof)$ THEN $PrioDiscardDataOof$ ELSE
IF ENABLED $(PrioDiscardAuthSsin)$ THEN $PrioDiscardAuthSsin$ ELSE

$\lor \exists\, ri \in Replid :$
$\quad \lor NextChangeFilter(ri)$
$\quad \lor NextChangeParent(ri)$

$\quad \lor NextCreateItem(ri)$
$\quad \lor NextUpdateVersion(ri)$

$\quad \lor NextMakeStar(ri)$
$\quad \lor NextMakeFreeisk(ri)$
$\quad \lor NextDensifyMw(ri)$
$\quad \lor NextDiscardDataOof(ri)$
$\quad \lor NextDiscardAuthSsin(ri)$

$\quad \lor NextInitiateSyncUptoParent(ri)$
$\quad \lor NextInitiateSyncFromParent(ri)$
$\quad \lor NextProcessMsg(ri)$

$\lor \exists\, tri,\, sri \in Replid \quad :$
$\quad \exists\, txisyes \in$ BOOLEAN $:$
$\quad NextRequestSync(tri,\, sri,\, txisyes)$ arbitrary topology sync

Conjoin the next state relation with a constraint on the new state. The constraint restricts the elaboration of the state space graph in order to enable model checking.

$Next \triangleq NextAny \land Constrain(state')$

INVARIANTS

The state is of the proper type.

$InvType \triangleq state \in State$

Every created version has either been superseded or is contained in some replica's store or in some sync data message.

$InvNoLoss \triangleq$
$\quad \forall\, tv \in state.truth :$
$\quad\quad \lor \exists\, tv1 \in state.truth : SupersedesVersion(tv1,\, tv)$
$\quad\quad \lor \exists\, ri \in Replid :$
$\quad\quad\quad$ LET
$\quad\quad\quad\quad r \triangleq GetReplicaFromState(ri,\, state)$
$\quad\quad\quad\quad vs \triangleq GetVersionsInReplica(r)$

IN
$\exists\, v \in vs : v.head.xi.vi = tv.head.xi.vi$

Every created version is contained in some replica's authoritative knowledge or in the authoritative knowledge of some sync data message.

$InvNoLossAuth \triangleq$
$\forall\, v \in state.truth :$
$\exists\, ri \in Replid :$
LET
$r \triangleq GetReplicaFromState(ri,\ state)$
IN
$\lor\ IsVersionInKn(v,\ r.authk)$
$\lor \exists\, msg \in GetRecvMsgsOfReplica(r) :$
$\land\ msg \in MsgSyncData$
$\land\ IsVersionInKn(v,\ msg.authk)$

All versions a replica has must be identical to true versions, except that its made-with knowledge can be a superset.

$InvStoreTruth \triangleq$
$\forall\, ri \in Replid :$
LET
$r \triangleq GetReplicaFromState(ri,\ state)$
IN
$\forall\, v \in GetVersionsInReplica(r) :$
$\exists\, tv \in state.truth :$
$\land\ [v \text{ EXCEPT } !.head.mw = \{\}] = [tv \text{ EXCEPT } !.head.mw = \{\}]$
$\land\ GeqKn(v.head.mw,\ tv.head.mw)$

All versions a replica has must have made-with knowledge in which each versid listed must either be (1) included in the made-with knowledge of the true version, (2) the versid of the version itself, or (3) the versid of a true version of a different item.

$InvStoreMw \triangleq$
$\forall\, ri \in Replid :$
LET
$r \triangleq GetReplicaFromState(ri,\ state)$
IN
$\forall\, v \in GetVersionsInReplica(r) :$
$\exists\, tv \in state.truth :$
$\land\ v.head.xi.vi = tv.head.xi.vi$
$\land \forall\, vi1 \in v.head.mw :$
$\lor\ vi1 \in tv.head.mw$
$\lor\ vi1 = v.head.xi.vi$
$\lor \exists\, tv1 \in state.truth :$
$\land\ tv1.head.xi.vi = vi1$
$\land\ tv1.head.xi.ii \neq v.head.xi.ii$

If a replica has v in its data store, it must have data knowledge of version v.

$$
\begin{array}{l}
InvKnowData \triangleq \\
\quad \forall\, ri \in Replid : \\
\quad \text{LET} \\
\quad\quad r \triangleq GetReplicaFromState(ri,\, state) \\
\quad \text{IN} \\
\quad \forall\, v \in r.datavs : IsVersionInIsetkn(v,\, r.dataisk)
\end{array}
$$

If a replica has v in its data store, it must not have data knowledge of any version that supersedes v.

$$
\begin{array}{l}
InvHaveDataSuperseder \triangleq \\
\quad \forall\, ri \in Replid : \\
\quad \text{LET} \\
\quad\quad r \triangleq GetReplicaFromState(ri,\, state) \\
\quad \text{IN} \\
\quad \forall\, v \in r.datavs : \\
\quad \forall\, tv \in state.truth : \\
\quad\quad SupersedesVersion(tv,\, v) \Rightarrow \neg IsVersionInIsetkn(tv,\, r.dataisk)
\end{array}
$$

If a replica has auth knowledge of v but does not have v in its auth store, it must have a superseder of v in its auth store.

$$
\begin{array}{l}
InvHaveAuthSuperseder \triangleq \\
\quad \forall\, ri \in Replid : \\
\quad \text{LET} \\
\quad\quad r \triangleq GetReplicaFromState(ri,\, state) \\
\\
\quad\quad miss(tv) \triangleq \\
\quad\quad\quad \land\, IsVersionInKn(tv,\, r.authk) \\
\quad\quad\quad \land\, \neg\exists\, v \in r.authvs : v.head.xi.vi = tv.head.xi.vi \\
\\
\quad \text{IN} \\
\quad \forall\, tv \in state.truth : miss(tv) \Rightarrow \\
\quad\quad \exists\, tv1 \in state.truth : \\
\quad\quad\quad \land\, SupersedesVersion(tv1,\, tv) \\
\quad\quad\quad \land\, \exists\, v1 \in r.authvs : v1.head.xi.vi = tv1.head.xi.vi
\end{array}
$$

If a replica knows version v, which is not superseded and which matches the replica's filter, then the replica must store v.

$$
\begin{array}{l}
InvDataFilter \triangleq \\
\quad \forall\, ri \in Replid : \\
\quad \text{LET} \\
\quad\quad r \triangleq GetReplicaFromState(ri,\, state) \\
\\
\quad\quad want(tv) \triangleq \land\, IsVersionInIsetkn(tv,\, r.dataisk) \\
\quad\quad\quad\quad\quad\quad \land\, \neg\exists\, tv1 \in state.truth : SupersedesVersion(tv1,\, tv) \\
\quad\quad\quad\quad\quad\quad \land\, IsVersionInFilter(tv,\, r.filter)
\end{array}
$$

$have(tv) \triangleq \land \exists\, v \in r.datavs : v.head.xi.vi = tv.head.xi.vi$

IN

$\forall\, tv \in state.truth : want(tv) \Rightarrow have(tv)$

If a replica has auth knowledge of version v, and version v is not superseded, then the replica has version v in its auth store.

$InvHaveAuth \triangleq$

$\forall\, ri \in Replid :$

LET

$r \triangleq GetReplicaFromState(ri,\ state)$

IN

$\forall\, vi \in r.authk :$

$\exists\, tv \in state.truth :$

$\land\ vi = tv.head.xi.vi$

$\land\ \lor \exists\, tv1 \in state.truth : SupersedesVersion(tv1,\ tv)$

$\lor \exists\, v \in r.authvs : v.head.xi.vi = tv.head.xi.vi$

If a replica has version v in its auth store, then the replica has auth knowledge of version v.

$InvKnowAuth \triangleq$

$\forall\, ri \in Replid :$

LET

$r \triangleq GetReplicaFromState(ri,\ state)$

IN

$\forall\, v \in r.authvs : IsVersionInKn(v,\ r.authk)$

TEMPORAL ASSUMPTIONS

Liveness assumption.

$Liveness \triangleq$

$\forall\, ri \in Replid :$

$\land\ \mathrm{WF}_{state}(NextInitiateSyncUptoParent(ri))$

$\land\ \mathrm{WF}_{state}(NextInitiateSyncFromParent(ri))$

$\land\ \mathrm{WF}_{state}(NextProcessMsg(ri))$

$\land\ \mathrm{WF}_{state}(NextMakeStar(ri))$

$\land\ \mathrm{WF}_{state}(NextMakeFreeisk(ri))$

$\land\ \mathrm{WF}_{state}(NextDensifyMw(ri))$

$\land\ \mathrm{WF}_{state}(NextDiscardDataOof(ri))$

Eventually we stop creating new versions.

$EventualAlwaysFrozenTruth \triangleq$

$\Diamond\Box(state.truth = state'.truth)$

Eventually we stop changing the tree.

$$
\begin{array}{l}
EventualAlwaysFrozenTree \triangleq \\
\quad \Diamond\Box \forall\, ri \in Replid : \\
\qquad \text{LET} \\
\qquad\quad r \triangleq GetReplicaFromState(ri,\, state) \\
\qquad\quad nr \triangleq GetReplicaFromState(ri,\, state') \\
\qquad \text{IN} \\
\qquad \wedge\, r.filter = nr.filter \\
\qquad \wedge\, r.parentri = nr.parentri
\end{array}
$$

Eventually we always have a proper tree.

$$
\begin{array}{l}
EventualAlwaysProperTree \triangleq \\
\quad \Diamond\Box IsProperTreeInState(state)
\end{array}
$$

TEMPORAL PROPERTIES

Filter consistency. Each replica stores exactly those non-superseded versions that match its filter.

$$
\begin{array}{l}
FilterConsistency \triangleq \\
\quad \forall\, ri \in Replid : \\
\quad \text{LET} \\
\qquad r \triangleq GetReplicaFromState(ri,\, state) \\
\qquad tvs \triangleq state.truth \\
\qquad ntvs \triangleq \{tv \in tvs : \neg\exists\, tv1 \in tvs : SupersedesVersion(tv1,\, tv)\} \\
\qquad fntvs \triangleq \{tv \in ntvs : IsVersionInFilter(tv,\, r.filter)\} \\
\quad \text{IN} \\
\qquad \{v.head.xi.vi : v \in r.datavs\} = \{tv.head.xi.vi : tv \in fntvs\}
\end{array}
$$

Eventual filter consistency.

$$AlwaysEventualFilterConsistency \triangleq \Box\Diamond FilterConsistency$$

Auth supersession. Each replica contains only non-superseded versions in its auth store.

$$
\begin{array}{l}
AuthSupersession \triangleq \\
\quad \forall\, ri \in Replid : \\
\quad \text{LET} \\
\qquad r \triangleq GetReplicaFromState(ri,\, state) \\
\quad \text{IN} \\
\quad \forall\, v \in r.authvs : \\
\qquad \neg\exists\, tv1 \in state.truth : SupersedesVersion(tv1,\, v)
\end{array}
$$

Eventual auth supersession.

$AlwaysEventualAuthSupersession \triangleq \Box\Diamond AuthSupersession$

Knowledge singularity. Each replica's data item-set knowledge is entirely star knowledge.

$$
\begin{array}{l}
KnSingularity \triangleq \\
\quad \forall\, ri \in Replid : \\
\quad \text{LET} \\
\qquad r \triangleq GetReplicaFromState(ri,\, state) \\
\quad \text{IN} \\
\qquad IsStarIsetkn(r.dataisk)
\end{array}
$$

Eventual knowledge singularity.

$AlwaysEventualKnSingularity \triangleq \Box\Diamond KnSingularity$

Made-with singularity. Among versions of items with no unresolved conflicts, the made-with knowledge is identical at all replicas.

$$
\begin{array}{l}
MwSingularity \triangleq \\
\quad \text{LET} \\
\qquad \text{Non-superseded true versions of itemid } ii. \\
\qquad ntvsii(ii) \triangleq \{tv \in state.truth : \\
\qquad\qquad \wedge\, tv.head.xi.ii = ii \\
\qquad\qquad \wedge\, \neg\exists\, tv1 \in state.truth : SupersedesVersion(tv1,\, tv) \\
\qquad\qquad \} \\
\qquad \text{There are no unresolved conflicts of itemid } ii. \\
\qquad freeii(ii) \triangleq \forall\, tv1,\, tv2 \in ntvsii(ii) : tv1 = tv2 \\
\qquad \text{All stored versions at all replicas.} \\
\qquad vs \triangleq \text{UNION}\, \{GetReplicaFromState(ri,\, state).datavs : ri \in Replid\} \\
\qquad \text{All stored versions of items with no unresolved conflicts.} \\
\qquad rvs \triangleq \{v \in vs : freeii(v.head.xi.ii)\} \\
\quad \text{IN} \\
\qquad \forall\, v1,\, v2 \in rvs : v1.head.mw = v2.head.mw
\end{array}
$$

Eventual made-with singularity.

$MwSingularity \triangleq \Box\Diamond MwSingularity$

VIEW

The *debug* component of the state contains commentary on the action that was used to arrive at the state. We disregard this commentary when comparing states for equality in the state graph.

$View \triangleq [state \text{ EXCEPT } !.debug = \{\}]$

SPECIFICATION

$Spec \triangleq$
- $\land\ Init$
- $\land\ \Box[Next]_{state}$
- $\land\ Liveness$
- $\land\ EventualAlwaysFrozenTruth$
- $\land\ EventualAlwaysFrozenTree$
- $\land\ EventualAlwaysProperTree$

THEOREM $Spec \Rightarrow$
- $\land\ \Box InvType$
- $\land\ \Box InvNoLoss$
- $\land\ \Box InvNoLossAuth$
- $\land\ \Box InvStoreTruth$
- $\land\ \Box InvStoreMw$
- $\land\ \Box InvKnowData$
- $\land\ \Box InvHaveDataSuperseder$
- $\land\ \Box InvHaveAuthSuperseder$
- $\land\ \Box InvDataFilter$
- $\land\ \Box InvHaveAuth$
- $\land\ \Box InvKnowAuth$
- $\land\ AlwaysEventualFilterConsistency$
- $\land\ AlwaysEventualAuthSupersession$
- $\land\ AlwaysEventualKnSingularity$
- $\land\ AlwaysEventualMwSingularity$

Appendix B

Model configurations

This chapter lists sample model configurations for the TLA+ specification listed in Appendix A. The first several configurations run to completion checking all invariants and temporal conditions with no errors found. The remaining configurations turn on various bugs and find counterexample model execution histories.

B.1 Model configuration ibx

```
CONSTANT Itemid = { i }
CONSTANT Replid = { a, b }
CONSTANT Content = { w, x }

CONSTANT NullReplid = nri

CONSTANT Parentinc = 1
CONSTANT MaxFilternum = 2
CONSTANT MaxParentnum = 1
CONSTANT MaxVersnum = 1
CONSTANT MaxSyncact = 1

CONSTANT MaxNzFilternum = 2
CONSTANT MaxNzParentnum = 1
CONSTANT MaxNzVersnum = 1
CONSTANT MaxNzSyncact = 1

CONSTANT MaxTotalFilternum = 4
CONSTANT MaxTotalParentnum = 1
CONSTANT MaxTotalVersnum = 1
CONSTANT MaxTotalSyncact = 1

SPECIFICATION Spec
VIEW View

INVARIANT InvType
INVARIANT InvNoLoss
INVARIANT InvNoLossAuth
INVARIANT InvStoreTruth
INVARIANT InvStoreMw
INVARIANT InvKnowData
INVARIANT InvHaveDataSuperseder
INVARIANT InvHaveAuthSuperseder
INVARIANT InvDataFilter
INVARIANT InvHaveAuth
INVARIANT InvKnowAuth

PROPERTY AlwaysEventualFilterConsistency
PROPERTY AlwaysEventualAuthSupersession
PROPERTY AlwaysEventualKnSingularity
PROPERTY AlwaysEventualMwSingularity
```

B.2 Model configuration icy

```
CONSTANT Itemid = { i }
CONSTANT Replid = { a, b, c }
CONSTANT Content = { w, x, y }

CONSTANT NullReplid = nri

CONSTANT Parentinc = 1
CONSTANT MaxFilternum = 1
CONSTANT MaxParentnum = 1
CONSTANT MaxVersnum = 1
CONSTANT MaxSyncact = 1

CONSTANT MaxNzFilternum = 1
CONSTANT MaxNzParentnum = 1
CONSTANT MaxNzVersnum = 1
CONSTANT MaxNzSyncact = 1

CONSTANT MaxTotalFilternum = 1
CONSTANT MaxTotalParentnum = 1
CONSTANT MaxTotalVersnum = 1
CONSTANT MaxTotalSyncact = 1

SPECIFICATION Spec
VIEW View

INVARIANT InvType
INVARIANT InvNoLoss
INVARIANT InvNoLossAuth
INVARIANT InvStoreTruth
INVARIANT InvStoreMw
INVARIANT InvKnowData
INVARIANT InvHaveDataSuperseder
INVARIANT InvHaveAuthSuperseder
INVARIANT InvDataFilter
INVARIANT InvHaveAuth
INVARIANT InvKnowAuth

PROPERTY AlwaysEventualFilterConsistency
PROPERTY AlwaysEventualAuthSupersession
PROPERTY AlwaysEventualKnSingularity
PROPERTY AlwaysEventualMwSingularity
```

B.3 Model configuration jbx

```
CONSTANT Itemid = { i, j }
CONSTANT Replid = { a, b }
CONSTANT Content = { w, x }

CONSTANT NullReplid = nri

CONSTANT Parentinc = 1
CONSTANT MaxFilternum = 2
CONSTANT MaxParentnum = 1
CONSTANT MaxVersnum = 1
CONSTANT MaxSyncact = 1

CONSTANT MaxNzFilternum = 2
CONSTANT MaxNzParentnum = 1
CONSTANT MaxNzVersnum = 1
CONSTANT MaxNzSyncact = 1

CONSTANT MaxTotalFilternum = 2
CONSTANT MaxTotalParentnum = 1
CONSTANT MaxTotalVersnum = 1
CONSTANT MaxTotalSyncact = 1

SPECIFICATION Spec
VIEW View

INVARIANT InvType
INVARIANT InvNoLoss
INVARIANT InvNoLossAuth
INVARIANT InvStoreTruth
INVARIANT InvStoreMw
INVARIANT InvKnowData
INVARIANT InvHaveDataSuperseder
INVARIANT InvHaveAuthSuperseder
INVARIANT InvDataFilter
INVARIANT InvHaveAuth
INVARIANT InvKnowAuth

PROPERTY AlwaysEventualFilterConsistency
PROPERTY AlwaysEventualAuthSupersession
PROPERTY AlwaysEventualKnSingularity
PROPERTY AlwaysEventualMwSingularity
```

B.4 Model configuration BugAuthBounceForever

```
CONSTANT Itemid = { i }
CONSTANT Replid = { a, b, c }
CONSTANT Content = { w }

CONSTANT NullReplid = nri

CONSTANT Parentinc = 1
CONSTANT MaxFilternum = 0
CONSTANT MaxParentnum = 0
CONSTANT MaxVersnum = 1
CONSTANT MaxSyncact = 1

CONSTANT MaxNzFilternum = 0
CONSTANT MaxNzParentnum = 0
CONSTANT MaxNzVersnum = 1
CONSTANT MaxNzSyncact = 1

CONSTANT MaxTotalFilternum = 0
CONSTANT MaxTotalParentnum = 0
CONSTANT MaxTotalVersnum = 1
CONSTANT MaxTotalSyncact = 1

SPECIFICATION Spec
VIEW View

INVARIANT InvType
INVARIANT InvNoLoss
INVARIANT InvNoLossAuth
INVARIANT InvStoreTruth
INVARIANT InvStoreMw
INVARIANT InvKnowData
INVARIANT InvHaveDataSuperseder
INVARIANT InvHaveAuthSuperseder
INVARIANT InvDataFilter
INVARIANT InvHaveAuth
INVARIANT InvKnowAuth

PROPERTY AlwaysEventualFilterConsistency

CONSTANT BugAuthBounceForever = TRUE
```

B.5 Model configuration BugContainFilter

```
CONSTANT Itemid = { i }
CONSTANT Replid = { a, b }
CONSTANT Content = { w, x }

CONSTANT NullReplid = nri

CONSTANT Parentinc = 1
CONSTANT MaxFilternum = 2
CONSTANT MaxParentnum = 0
CONSTANT MaxVersnum = 1
CONSTANT MaxSyncact = 1

CONSTANT MaxNzFilternum = 1
CONSTANT MaxNzParentnum = 0
CONSTANT MaxNzVersnum = 1
CONSTANT MaxNzSyncact = 1

CONSTANT MaxTotalFilternum = 2
CONSTANT MaxTotalParentnum = 0
CONSTANT MaxTotalVersnum = 1
CONSTANT MaxTotalSyncact = 1

SPECIFICATION Spec
VIEW View

INVARIANT InvType
INVARIANT InvNoLoss
INVARIANT InvNoLossAuth
INVARIANT InvStoreTruth
INVARIANT InvStoreMw
INVARIANT InvKnowData
INVARIANT InvHaveDataSuperseder
INVARIANT InvHaveAuthSuperseder
INVARIANT InvDataFilter
INVARIANT InvHaveAuth
INVARIANT InvKnowAuth


CONSTANT BugContainFilter = TRUE
```

B.6 Model configuration BugLearnSend

```
CONSTANT Itemid = { i }
CONSTANT Replid = { a, b, c }
CONSTANT Content = { w, x }

CONSTANT NullReplid = nri

CONSTANT Parentinc = 1
CONSTANT MaxFilternum = 0
CONSTANT MaxParentnum = 0
CONSTANT MaxVersnum = 1
CONSTANT MaxSyncact = 1

CONSTANT MaxNzFilternum = 0
CONSTANT MaxNzParentnum = 0
CONSTANT MaxNzVersnum = 2
CONSTANT MaxNzSyncact = 1

CONSTANT MaxTotalFilternum = 0
CONSTANT MaxTotalParentnum = 0
CONSTANT MaxTotalVersnum = 2
CONSTANT MaxTotalSyncact = 1

SPECIFICATION Spec
VIEW View

INVARIANT InvType
INVARIANT InvNoLoss
INVARIANT InvNoLossAuth
INVARIANT InvStoreTruth
INVARIANT InvStoreMw
INVARIANT InvKnowData
INVARIANT InvHaveDataSuperseder
INVARIANT InvHaveAuthSuperseder
INVARIANT InvDataFilter
INVARIANT InvHaveAuth
INVARIANT InvKnowAuth


CONSTANT BugLearnSend = TRUE
```

B.7 Model configuration BugLearnStore

```
CONSTANT Itemid = { i }
CONSTANT Replid = { a, b, c }
CONSTANT Content = { w, x }

CONSTANT NullReplid = nri

CONSTANT Parentinc = 1
CONSTANT MaxFilternum = 0
CONSTANT MaxParentnum = 0
CONSTANT MaxVersnum = 1
CONSTANT MaxSyncact = 1

CONSTANT MaxNzFilternum = 0
CONSTANT MaxNzParentnum = 0
CONSTANT MaxNzVersnum = 2
CONSTANT MaxNzSyncact = 1

CONSTANT MaxTotalFilternum = 0
CONSTANT MaxTotalParentnum = 0
CONSTANT MaxTotalVersnum = 2
CONSTANT MaxTotalSyncact = 1

SPECIFICATION Spec
VIEW View

INVARIANT InvType
INVARIANT InvNoLoss
INVARIANT InvNoLossAuth
INVARIANT InvStoreTruth
INVARIANT InvStoreMw
INVARIANT InvKnowData
INVARIANT InvHaveDataSuperseder
INVARIANT InvHaveAuthSuperseder
INVARIANT InvDataFilter
INVARIANT InvHaveAuth
INVARIANT InvKnowAuth


CONSTANT BugLearnStore = TRUE
```

B.8 Model configuration BugOmitDiscardAuthSsin

```
CONSTANT Itemid = { i }
CONSTANT Replid = { a, b }
CONSTANT Content = { w }

CONSTANT NullReplid = nri

CONSTANT Parentinc = 1
CONSTANT MaxFilternum = 0
CONSTANT MaxParentnum = 0
CONSTANT MaxVersnum = 1
CONSTANT MaxSyncact = 1

CONSTANT MaxNzFilternum = 0
CONSTANT MaxNzParentnum = 0
CONSTANT MaxNzVersnum = 2
CONSTANT MaxNzSyncact = 1

CONSTANT MaxTotalFilternum = 0
CONSTANT MaxTotalParentnum = 0
CONSTANT MaxTotalVersnum = 2
CONSTANT MaxTotalSyncact = 1

SPECIFICATION Spec
VIEW View

INVARIANT InvType
INVARIANT InvNoLoss
INVARIANT InvNoLossAuth
INVARIANT InvStoreTruth
INVARIANT InvStoreMw
INVARIANT InvKnowData
INVARIANT InvHaveDataSuperseder
INVARIANT InvHaveAuthSuperseder
INVARIANT InvDataFilter
INVARIANT InvHaveAuth
INVARIANT InvKnowAuth

PROPERTY AlwaysEventualAuthSupersession

CONSTANT BugOmitDiscardAuthSsin = TRUE
```

B.9 Model configuration BugOmitDiscardDataOof

```
CONSTANT Itemid = { i }
CONSTANT Replid = { a, b }
CONSTANT Content = { w, x }

CONSTANT NullReplid = nri

CONSTANT Parentinc = 1
CONSTANT MaxFilternum = 0
CONSTANT MaxParentnum = 0
CONSTANT MaxVersnum = 1
CONSTANT MaxSyncact = 1

CONSTANT MaxNzFilternum = 0
CONSTANT MaxNzParentnum = 0
CONSTANT MaxNzVersnum = 2
CONSTANT MaxNzSyncact = 1

CONSTANT MaxTotalFilternum = 0
CONSTANT MaxTotalParentnum = 0
CONSTANT MaxTotalVersnum = 2
CONSTANT MaxTotalSyncact = 1

SPECIFICATION Spec
VIEW View

INVARIANT InvType
INVARIANT InvNoLoss
INVARIANT InvNoLossAuth
INVARIANT InvStoreTruth
INVARIANT InvStoreMw
INVARIANT InvKnowData
INVARIANT InvHaveDataSuperseder
INVARIANT InvHaveAuthSuperseder
INVARIANT InvDataFilter
INVARIANT InvHaveAuth
INVARIANT InvKnowAuth

PROPERTY AlwaysEventualFilterConsistency

CONSTANT BugOmitDiscardDataOof = TRUE
```

B.10 Model configuration BugOmitIndMoveouts

```
CONSTANT Itemid = { i }
CONSTANT Replid = { a, b, c }
CONSTANT Content = { w, x }

CONSTANT NullReplid = nri

CONSTANT Parentinc = 1
CONSTANT MaxFilternum = 0
CONSTANT MaxParentnum = 0
CONSTANT MaxVersnum = 1
CONSTANT MaxSyncact = 1

CONSTANT MaxNzFilternum = 0
CONSTANT MaxNzParentnum = 0
CONSTANT MaxNzVersnum = 2
CONSTANT MaxNzSyncact = 1

CONSTANT MaxTotalFilternum = 0
CONSTANT MaxTotalParentnum = 0
CONSTANT MaxTotalVersnum = 2
CONSTANT MaxTotalSyncact = 1

SPECIFICATION Spec
VIEW View

INVARIANT InvType
INVARIANT InvNoLoss
INVARIANT InvNoLossAuth
INVARIANT InvStoreTruth
INVARIANT InvStoreMw
INVARIANT InvKnowData
INVARIANT InvHaveDataSuperseder
INVARIANT InvHaveAuthSuperseder
INVARIANT InvDataFilter
INVARIANT InvHaveAuth
INVARIANT InvKnowAuth

PROPERTY AlwaysEventualFilterConsistency

CONSTANT BugOmitIndMoveouts = TRUE
```

B.11 Model configuration BugOmitMoveouts

```
CONSTANT Itemid = { i }
CONSTANT Replid = { a, b }
CONSTANT Content = { w, x }

CONSTANT NullReplid = nri

CONSTANT Parentinc = 1
CONSTANT MaxFilternum = 0
CONSTANT MaxParentnum = 0
CONSTANT MaxVersnum = 1
CONSTANT MaxSyncact = 1

CONSTANT MaxNzFilternum = 0
CONSTANT MaxNzParentnum = 0
CONSTANT MaxNzVersnum = 2
CONSTANT MaxNzSyncact = 1

CONSTANT MaxTotalFilternum = 0
CONSTANT MaxTotalParentnum = 0
CONSTANT MaxTotalVersnum = 2
CONSTANT MaxTotalSyncact = 1

SPECIFICATION Spec
VIEW View

INVARIANT InvType
INVARIANT InvNoLoss
INVARIANT InvNoLossAuth
INVARIANT InvStoreTruth
INVARIANT InvStoreMw
INVARIANT InvKnowData
INVARIANT InvHaveDataSuperseder
INVARIANT InvHaveAuthSuperseder
INVARIANT InvDataFilter
INVARIANT InvHaveAuth
INVARIANT InvKnowAuth

PROPERTY AlwaysEventualFilterConsistency

CONSTANT BugOmitMoveouts = TRUE
```

B.12 Model configuration BugOmitRebuildOnUnshrink

```
CONSTANT Itemid = { i }
CONSTANT Replid = { a, b }
CONSTANT Content = { w }

CONSTANT NullReplid = nri

CONSTANT Parentinc = 1
CONSTANT MaxFilternum = 1
CONSTANT MaxParentnum = 0
CONSTANT MaxVersnum = 1
CONSTANT MaxSyncact = 1

CONSTANT MaxNzFilternum = 1
CONSTANT MaxNzParentnum = 0
CONSTANT MaxNzVersnum = 1
CONSTANT MaxNzSyncact = 1

CONSTANT MaxTotalFilternum = 1
CONSTANT MaxTotalParentnum = 0
CONSTANT MaxTotalVersnum = 1
CONSTANT MaxTotalSyncact = 1

SPECIFICATION Spec
VIEW View

INVARIANT InvType
INVARIANT InvNoLoss
INVARIANT InvNoLossAuth
INVARIANT InvStoreTruth
INVARIANT InvStoreMw
INVARIANT InvKnowData
INVARIANT InvHaveDataSuperseder
INVARIANT InvHaveAuthSuperseder
INVARIANT InvDataFilter
INVARIANT InvHaveAuth
INVARIANT InvKnowAuth


CONSTANT BugOmitRebuildOnUnshrink = TRUE
```

B.13 Model configuration BugUnionFreeisk

```
CONSTANT Itemid = { i }
CONSTANT Replid = { a, b }
CONSTANT Content = { w }

CONSTANT NullReplid = nri

CONSTANT Parentinc = 1
CONSTANT MaxFilternum = 0
CONSTANT MaxParentnum = 0
CONSTANT MaxVersnum = 1
CONSTANT MaxSyncact = 1

CONSTANT MaxNzFilternum = 0
CONSTANT MaxNzParentnum = 0
CONSTANT MaxNzVersnum = 2
CONSTANT MaxNzSyncact = 1

CONSTANT MaxTotalFilternum = 0
CONSTANT MaxTotalParentnum = 0
CONSTANT MaxTotalVersnum = 2
CONSTANT MaxTotalSyncact = 1

SPECIFICATION Spec
VIEW View

INVARIANT InvType
INVARIANT InvNoLoss
INVARIANT InvNoLossAuth
INVARIANT InvStoreTruth
INVARIANT InvStoreMw
INVARIANT InvKnowData
INVARIANT InvHaveDataSuperseder
INVARIANT InvHaveAuthSuperseder
INVARIANT InvDataFilter
INVARIANT InvHaveAuth
INVARIANT InvKnowAuth


CONSTANT BugUnionFreeisk = TRUE
```

B.14 Model configuration BugUnshrinkLearn

```
CONSTANT Itemid = { i }
CONSTANT Replid = { a, b }
CONSTANT Content = { w, x }

CONSTANT NullReplid = nri

CONSTANT Parentinc = 1
CONSTANT MaxFilternum = 2
CONSTANT MaxParentnum = 0
CONSTANT MaxVersnum = 1
CONSTANT MaxSyncact = 1

CONSTANT MaxNzFilternum = 1
CONSTANT MaxNzParentnum = 0
CONSTANT MaxNzVersnum = 1
CONSTANT MaxNzSyncact = 1

CONSTANT MaxTotalFilternum = 2
CONSTANT MaxTotalParentnum = 0
CONSTANT MaxTotalVersnum = 1
CONSTANT MaxTotalSyncact = 1

SPECIFICATION Spec
VIEW View

INVARIANT InvType
INVARIANT InvNoLoss
INVARIANT InvNoLossAuth
INVARIANT InvStoreTruth
INVARIANT InvStoreMw
INVARIANT InvKnowData
INVARIANT InvHaveDataSuperseder
INVARIANT InvHaveAuthSuperseder
INVARIANT InvDataFilter
INVARIANT InvHaveAuth
INVARIANT InvKnowAuth


CONSTANT BugUnshrinkLearn = TRUE
```

B.15 Model configuration BugUnshrinkMoveout

```
CONSTANT Itemid = { i }
CONSTANT Replid = { a, b }
CONSTANT Content = { w, x }

CONSTANT NullReplid = nri

CONSTANT Parentinc = 1
CONSTANT MaxFilternum = 1
CONSTANT MaxParentnum = 0
CONSTANT MaxVersnum = 1
CONSTANT MaxSyncact = 1

CONSTANT MaxNzFilternum = 1
CONSTANT MaxNzParentnum = 0
CONSTANT MaxNzVersnum = 2
CONSTANT MaxNzSyncact = 1

CONSTANT MaxTotalFilternum = 1
CONSTANT MaxTotalParentnum = 0
CONSTANT MaxTotalVersnum = 2
CONSTANT MaxTotalSyncact = 1

SPECIFICATION Spec
VIEW View

INVARIANT InvType
INVARIANT InvNoLoss
INVARIANT InvNoLossAuth
INVARIANT InvStoreTruth
INVARIANT InvStoreMw
INVARIANT InvKnowData
INVARIANT InvHaveDataSuperseder
INVARIANT InvHaveAuthSuperseder
INVARIANT InvDataFilter
INVARIANT InvHaveAuth
INVARIANT InvKnowAuth


CONSTANT BugUnshrinkMoveout = TRUE
```

Index