Unitraverse Client Application
Data Safety

by

Bradley A. Pliam

THE BASIC CLIENT APP VAULT IMPLEMENTATION

The client application maintains a vault of information, holding a user's links and other resources in a tree structure. Each structural node of the tree has a parent (generally one) and child nodes, as well as other data that belongs to each node. A node is basically a software object, in memory, that can be associated with other nodes, with parents and children, and that can contain other types of information as well. A non-structural node is just a "leaf" node. It has no children and appears at the lowest level of every branch of the tree.

In order to support symbolic link nodes, we create placeholders for another node that will be brought in to some leaf location at a later point. The placeholder is a stub node, and the node being referenced is called a guest or target. This means a process is required to resolve the links. The tree starts out as an unexpanded tree, which contains link stubs here and there, but then it is expanded. This means the objects that the link stub refers to will be made to reside at that location in the tree instead of the stub node. This guest node is not a copy of an original node, it is actually the original node object. There are obvious space-saving reasons to implement a tree this way.

A consequence of having a guest (or 'target') node exist in multiple places, in the structure of an expanded tree, is that when we examine such a target node, there is not a simple way to tell that it is a guest object replacing a stub, or if it is a target object that has resided there all along. There needs to be bookkeeping information that is maintained to identify what nodes are at "link" locations, and what nodes are at "non-link" locations. The discussion below talks about 'PCI' information, which is what we use to describe a *class* of locations in order to make this determination.

EDITS THAT CAUSE STRUCTURAL CHANGES

Some edits, for example, changing the title of some node, has no impact on the structure of the vault. There are however, edits that do affect the vault-tree structure, such as adding a new node, or removing a child node.

It is crucial to pay close attention to where structural changes in the client app codebase because this affects the information that is vital to replacing stubs. When the user saves any changes, the list of stubs needs to get swapped back into their proper location, and the guest objects are swapped out, since the guest objects must not be saved as guests in the vault, but their link stubs must be in place and get saved to disk. This swapping procedure will need to be reliable in replacing the correct object, the one that is the intended guest, with the correct link stubs. The danger that must be averted is that an invalid swap could conceivably be caused by poorly implemented structural edits that unwittingly replace the wrong branches with stubs. Of course, this basically removes content from the vault that should not be removed… in short, causing the loss of data. The discussion that follows will highlight the precautions that have been taken (as of version 1.2.141, with the exception of step #7), and that are being recommended to end-users, to assure that this does not happen.

DATA SAFETY STRATEGIES SUMMARIZED

The key strategies to prevent data loss from happening because of a structural edit, are the following:

1) Assure correctness of code design - this document discusses the initial steps taken toward this end
2) Automated testing to detect before-and-after correctness for operations that affect user data, on a case-by-case basis
3) Runtime checks that are done after every edit operation. (This is a non-optional check that does not include app state validation that can be turned on and off in the user settings.json file.)
4) A dynamically updated log file and real-time debug console output to see what happens in real world use situations
5) A 'blockingSaveOp' switch to prevent the user from saving (locally or to the cloud) if something is not correct in the state of the running application, or if something didn't go right during an edit operation which also endangers correctness in the data. This cannot be turned off in the settings.
6) As a redundant safeguard, implement a checking method before save that runs through each swap record, verifying that the index positions, parent and child of every record matches the current tree configuration. The save operations are blocked if this check fails.
7) As a redundant safeguard, build in detection of branch manipulations and edits that have been made outside of the API that is provided for structural vault edits.
8) As a redundant safeguard, advise any users to scan their applet code for any direct manipulations of nodes.
9) Give a conspicuous recommendation to all users that they back-up their data, so that it can be rolled back if something still goes wrong

Additional strategies to thwart malware and trojan horse attacks:

10) Work behind a firewall
11) Use MD5 to verify any downloaded version of the application (currently not available).
12) Have updated virus protection running on client systems

EDITS THAT CAUSE NON-DESTRUCTIVE STRUCTURAL CHANGES

When adding a node, this causes a structural change. It is non-destructive because we are not removing anything, but rather adding something new. With this type of edit, though, there is still the important aspect of making sure that the bookkeeping is correctly updated and maintained for stub-swapping. The code regions that will potentially affect the correctness of the edit and of the bookkeeping mechanisms can fail, for many reasons. To safeguard against having this become a source of data loss, we set a flag upon entering such a region of code. The "blockingSaveOp" flag will prevent any future save operations from happening, and should the code that makes structural modifications fail during execution, that flag will remain set. Only when the code completes successfully do we replace the most restrictive of either: a) the flag value before entering the critical region or b) the current alert flag indicating if there was a problem when making the update.

The non-destructive edit operations will require that special attention is paid to whether the edit operation completes, and if it causes errors and warnings that pertain to the integrity of the vault. These are specifically code regions that will:

1) add a node in the standard way
2) add nodes from pasted JSON information
3) add a node from an applet

As execution passes to the client application there are some non-hazardous setup code statements that will precede a critical region of code that starts to implement changes to the application state. The regions that are critical are easy to identify and have highly noticeable comments to alert future programmers of their importance. (Currently the critical code section is made up of those statements starting with an auxiliary method that collapses the vault tree. After the edit is complete, there will be some updates provided to the bookkeeping data structures that are mentioned above which have critical importance to data safety and integrity. Finally, the vault tree will be expanded using new information. This marks the end of the critical section for all non-destructive edits.)

This critical code region and the things that need to happen in that region are the reason we implement a 'blockingSaveOp' flag that remains set if the region doesn't complete cleanly. These are also the reason we prohibit direct access to tree objects by applets.

EDITS THAT CAUSE DESTRUCTIVE STRUCTURAL CHANGES

When removing a node, a destructive change occurs, causing the tree to be structurally different than it was previously. When a tree supports symlinks, and also when a user path is allowed to contain cycles, we need to alter the tree structure so that it does not depend on any removed elements, and we may from time to time need to re-establish a current user location. From a data-loss standpoint, the destructive 'remove' operation safety considerations and the safety aspects of the required bookkeeping are not that much different from those of the non-destructive operations. The discussion below will try to make that clear.

The description of the critical code region for the 'non-destructive' edits is nearly the same for the destructive 'remove' operation code. This critical 'remove' code must perform many more updates to bookkeeping and sometimes implement more complex changes to the tree. It is still, however, equally as simple to isolate and pay special attention to code lines that comprise a critical region for 'remove' operations. The same 'blockingSaveOp' safeguard can be implemented just as easily, even though there will be more places to flag things that constitute a detrimental condition.

THE REMOVE OPERATION

The Unitraverse client application allows users to delete or 'remove' any node in their vault. When an item is selected, clicking 'remove' will attempt to remove the item, but users will be given the chance to back out if the operation breaks links. There is a modified behavior for 'remove' when the 'ALT' key is held down while 'remove' is clicked. 'ALT-remove' only removes the currently selected item from it's parent (and from any and all instances of that parent), but, with 'ALT-remove', if that selected item has children, these children become installed directly under the parent at precisely the location where the deleted item had previously resided.

The 'remove' operation changes the structure of the vault in a destructive way. This makes 'remove' different than other operations, because it forces us to deal with dangling links, those links, in other words, left behind to point to a part of the tree that is now missing. Also we do not want the current user navigation path location to point to something that no longer exists.

The basic algorithm for determining if there are any dangling, unresolvable links after a 'remove', requires being able to compile a list of all globally indigenous or unmediated pre-expansion locations that become, or are identified as victim locations. It must be made clear that by "location" we mean a specific Parent-Child-ChildIndex (PCI) combination which will uniquely identify a tree location prior to engrafting guest objects into a tree, in other words prior to "expanding" a tree. This PCI location may equate to a set of 'absolute' expanded tree locations.

Absolute locations are conceptually important here because they are what would define any location of a tree, even after tree expansion. It might be helpful to think of the absolute location as being pretty much like an absolute file system location, and the PCI as a very short relative file system location. This file system parallel can be drawn even further, because, as directory paths have names that might repeat, and still be comprised of objects that do not recur, so also this tree can have nodes with titles or labels that are repetitious and yet still have node objects with entirely unique addresses. Also, as we have mentioned already, the symbolic links in the client app are very much like those in the file system world. All this was worth mentioning so that we can make the following points:

Whereas the PCI location is entirely unambiguous only prior to the expansion of the tree, the absolute location is something that would disambiguate locations after expansion. However, it is the PCI that is being used for calculating victims and dangling pointers, because the practical concern for severing or creating connections is adequately addressed by the PCI and what it represents. The absolute location is not needed. We care only that connections are severed or created for purposes of the unexpanded tree, and leave it to other code to create the expanded tree as a downstream consequence of connection changes we make. Looking at only the PCI is abstracting away the full path to some node location. The usefulness of this PCI information is undiminished in an expanded tree because a 'remove' operation will involve the connection being referred to by a current location in the tree that provides the PCI. The connection can be severed and when the tree is redrawn, the specifics of the broken connection will faithfully be reproduced, in the case that it is replicated in the redraw process.

Pre-expansion locations existed before any guest objects were engrafted into the tree, and they survive after expansion also. Once a tree is expanded, the pre-expansion location values are often no longer able to uniquely identify an absolute tree location, but they still do uniquely identify a class of parent-child connections. These PCI values now can be potentially found to exist at any number of derived locations after expansion. While the combination of parent-childIndex does not always disambiguate every post-expansion tree location, it does reliably provide a unique signature for a pre-expansion relationship or connection, even if that relationship exists in multiple locations of an expanded tree. Tracking of PCIs is what allows the system to determine if something is a symlink or not… if some PCI is not a member of the group of pre-expansion PCIs we assume that it is a link that was created during expansion. There may be any number of nodes hosting the same guest, and the PCI will help identify a group of connections that involve the same parent and child at some particular index. A severing of one node in such a group will severe all other nodes in that group.

For even more clarity, given a tree 'T' comprised of a set of objects 'S', such that 'S' = {'A','B','C'}, so that 'T' contains 1 and only 1 'A' object, 1 and only 1 'B' object, etc…. Then, for an expanded tree 'T(x)' that is based on 'T', where elements of 'S' are used any number of times to build 'T(x)', the changes to any element in 'S' will propagate throughout both 'T(x)' as well as throughout 'T'. We can see that the creation process that produced 'T' using some version of 'A', and that used to derive 'T(x)', need to similarly and reliably produce their 'T' and 'T(x)' using any alternate 'A', so, using 'A(1)' instead of 'A(0)'. This is relevant because 'A(0)' and 'A(1)' can be considered to be different versions of some parent node 'A', where 'A(0)' has some node 'C' as one of it's children, and 'A(1)' does not have 'C' as a child. Redrawing the tree means that where there were instances of 'A(0)' formerly, now the 'A(1)' instances will be used. The process itself may not realize that it's using 'A(0)' or 'A(1)', it just knows to draw using the current 'A', but the consequences of that 'A' being 'A(1)' will be seen in the end result. When 'A(1)' is used, the removed object that might have been a descendant of 'A', no longer is there. What descendants are present or absent is simply the consequence of a process that is unconcerned with what happens at deeper levels.

This should be an intuitive backdrop that users will be able to somewhat understand when they do a 'remove' operation. Some users will also have the benefit of being familiar with symbolic links in a file system. They fully expect that sometimes when you delete a directory, you are deleting the original directory, but other times you are just deleting a pointer to the original, in other words deleting the link to a directory. In our client app system, we allow the deletion of prototypical objects, even if that will leave dangling links that still make reference to it.

CLARIFYING TERMINOLOGY

The terms "original node" and "indigenous node" are somewhat vague. All JavaScript objects that comprise a vault are originally incorporated into a collapsed tree (i.e,. they are all members of set 'S' above), but the expanded tree locations of any object will be determinative of whether they are guest objects, or an object that originally existed at some child position under a particular parent. An "original object" then is unambiguously understood as a node that has always existed at some defining position among the children of some particular parent object.

THE IMPORTANCE OF UPDATING AN ENGRAFTED OBJECTS LIST FROM AN ACCURATE VICTIM LIST

From a data integrity standpoint some implementation detail around 'remove' will be quite important. The list used to swap stub objects with target objects must be updated when edits occur and the 'remove' operation must determine specifically what PPC locations are being removed and whether locations are that of the primary

object or that of a victim in the victim list. Correct calculation and maintenance of these structures is fundamental to making changes to user data in a reliable way. If some entry in the swap list ('engraftedObjects') is missing when it should be present, or if some entry is present when it was supposed to have been removed, the clear danger is unintentional removal of data from the vault.

A user will generally see immediate results from the edits they make, as each operation will update the in-memory tree and display a new page based on that updated tree. Some display arrangements show changes more fully than others. None-the-less, awareness of details around victim analysis can demonstrate achievability and difficulty. Of course, it is not left to users to detect whether something has gone wrong internally or not.

A CASCADE OF VICTIMS

There will be, obviously, at least one primary victim as the result of any successful 'remove' operation, but removals can also have repercussions for other nodes and the changes can reverberate to other locations in an expanded tree. Any redrawn tree should exhibit the consequences of the 'remove' operation at any location where some particular PCI has been deleted.

The effect of a 'remove' operation can be huge, as the victims are often the children and other descendants of the removed item. So long as the 'ALT' key was not pressed, in other words, with a basic 'remove' operation, there may be deeper descendants removed as a consequence, causing a continued cascade of victims, which can extend down into all branches of a primary (selected) node. When a symlink is encountered among the descendants of a victim, it's location and the associated stub objects are treated as a victim, but the guest object is not designated as a victim, at least not from that particular cascade and not by virtue of it residing at any other guest locations marked for deletion. The presence of a symlink prevents the process from looking further for victims among deeper descendants of that location, since it is the location of a guest. Nodes attached by the link engrafting mechanism will have needed to have some original location in a non-expanded tree. If no such original location exists, it cannot be incorporated as a guest object, which implies that every object that is a guest in some location has some other original, indigenous location to which it belongs. That indigenous location is the basis for excluding all other guest locations as a means to add some node to the list of victims.

DETERMINING WHICH OBJECTS ARE GUESTS

It is essential that we are able to differentiate instances of objects that are guests, from those that are in their indigenous locations.

Before an object can be a guest, it must be targetable and given an 'id'. A target node is one that has an 'id' field. But the presence of an 'id' field is not enough, the need for internal tracking of guests and non-duplicate targets is unavoidable. The tracking records will contain PCI that gets associated with a child object in order to make many of the determinations the system needs to make. There is a repertoire of checks and methods built for this system that can help determine if the node location being encountered is that of a symlink, a duplicate target, an original target or just a nondescript node. The details of this are omitted here but in general: the start-up routines will capture the locations of target stubs having a 'targetId' field, and target nodes having an 'id' field. The PCI details are also associated with the specialized nodes and these are collected into data structures that can be used to quickly determine when a node is being used as a symlink guest.

As nodes are added or removed, this tracking information can be updated in a fairly straightforward way. The updates made to all bookkeeping data structures will need to implement changes to PCI information in a reliable way. The most challenging aspect is that multiple 'remove' victims can send bump requests that have already been sent for a particular edit. Therefore, bumps that take place from adding or removing sibling nodes are implemented via a PCI registry. This allows there to be reliable updates to all data structures which "subscribe" to a PCI entry, without bumping up or down too many or too few times. Automated testing verifies that this is done reliably.

ADDED COMPLEXITY FROM THE 'ALT' MODIFIER KEY FOR 'REMOVE'

Added complexity will come from the unique behavior of the 'alt' key. The most simple type of 'remove' operation deletes an entire branch represented by the currently selected child label. The default use of the basic 'remove' command will delete the selected label (percept), it's corresponding child node, if any, and also remove any and all of it's descendants. If the 'alt' modifier key is pressed when issuing the 'remove' command, only the selected node is deleted, and the rest of the branch under that selected node will be inserted back into the location of the removed node.

The 'alt' key being pressed means that the descendants of a victim node will still be around, or at least, generally be unaffected by the 'remove' in cases where the descendants are not recursive instances of the primary victim. More details on how this plays out will follow.

ADDED COMPLEXITY ARISING FROM DUPLICATE IDS

The system does not allow vaults that have different objects sharing the same 'id' field to be saved to permanent storage. The paste and save -> from JSON operations allow the introduction of such duplicates into the vault, but they are flagged immediately for the end user. These duplicates are tracked and can be modified as a group in order that the vault can then be saved. When the vault does contain duplicate ids, no symlinks are created from them and they are kept from having an affect on internal tree iteration, which does depend on the 'id' field and target membership in the tgtIdTgtMap.

If by some unconventional actions, duplicate ids are introduced into a vault outside of the client app, this gives rise to the possibility that the first encountered instance of the non-unique id becomes the target, even if that was not intended or not the original configuration. While this situation would actually affect tree iteration, as infinite loops get avoided because of the presence of target ids, the affect is benign. It simply means that any recognized target guest, the ones that are allowed to be replicated multiple times in the expanded tree, will be visited one and only one time, while the nodes with a duplicate target id will be treated as regular nodes having no unintentional effect on tree iteration. Since the latter are not copied to multiple locations through the link mechanism, this is ok. The strategy requires that iteration is not simply based on encountering an 'id' field, but also based on membership in the tgtIdTgtMap.

ADDED COMPLEXITY FROM LINKS TO EXTERNAL VAULTS FOR 'REMOVE'

Another complicating factor that will not be as important right away is if the links bring in branches from another vault. The source of these external branches will not be affected by 'remove' operations in a host vault. Attempts to 'remove' a guest object from a parent that lives in some other vault will be blocked unless that guest object is native to the current vault, implying that it has been navigated to via some means of nested linking between vaults.

ADDED COMPLEXITY FROM CYCLICAL RECURSION FOR 'REMOVE'

When we introduce the concept of links (more-or-less things that are like symlinks in a file system), there is a potential for endless loops in a cycle. When linking allows some descendent to harbor one of its own ancestor nodes as a guest, a cycle is created. This introduces problems of software code that iterates through an assembled tree, where it is possible to iterate repeatedly over a cycle, which will produce a "stack overflow" error if the recursive descent is allowed to continue indefinitely. When users navigate a tree that contains such a cycle, they will see the cyclical pattern showing in the navigation path as they continue further and further within the cycle. If users did this enough times, they would hit the same "stack overflow" issue. It's merely boredom that prevents this from happening.

The presence of cycles does somewhat affect the complexity of a 'remove' operation, because a single 'remove' can invalidate the current path, or potential successors if the 'alt' key is pressed. The following strategy for determining dangling links and implementing an 'alt' move operation, talks about how these details are handled.

CASE BY CASE SUMMARY OF THE EFFECT AND BASIC ALGORITHMS OF 'REMOVE'

The presence of symlinks will sometimes, but not always, impact the removal operation. Each of the following things can be links: 1) the parent or scopeItem, 2) the

primary target being removed, 3) a descendant of the primary target, or 4) an ancestor node found in the path leading to the current scope item. This gives us 8 cases to consider:

Case 1a - a scope item is not a link
Case 1b - the scope item is a link
Case 2a - the primary victim is not a link
Case 2b - the primary victim is a link
Case 3a - a descendant of a primary victim is not a link
Case 3b - a descendant of a primary victim is a link
Case 4a - an ancestor of the current scope item is not a link
Case 4b - an ancestor of the current scope item is a link

For both Case # 1a and 1b: whether or not the scope item is a link does not matter for our present purpose. When the scope item is not a link, it presents no special circumstance for app behavior or finding victims. Likewise, when the scopeItem is a link node, there is nothing special to consider. It is entirely unimportant to app behavior for 'remove' and for calculating victims. The scope item is more-or-less just a window or stage from which a user is able to access and interact with some particular set of child objects, one of which might be selected. If the current path must be reset after removing some node, the ancestors become important, but, even then, it will not matter if they are links or not…what matters then is that they will still exist within the redrawn tree.

For all the following 2a and 2b cases, the primary victim, the item currently selected at the current scope, is no longer available at the selected child position under the current scope item. The direct connection from the scope item to that victim is severed and gone unless, for some reason, a redundant connection exists at some other child index position. For case 2a, the primary victim ceases to exist anywhere, since it will be the last remaining instance to be removed if the 'remove' is allowed, whereas for case 2b, the primary victim, since it was accessed and manipulated through the link mechanism, being that is was only a link node, would continue to reside at other locations in the tree.

Case # 2a: When the primary victim is not a link, there are 4 cases within this overall case:
Case i - the 'alt' key is not pressed
Case ii - the 'alt' key is pressed and the direct grandchild is not a link
Case iii - the 'alt' key is pressed and the direct grandchild is a link

When the 'alt' key is not pressed in a 2a or 2b context, that will mean that, in addition to the primary victim being removed, its children will also be severed and gone in terms of one of its PCI locations. All the descendant nodes of the primary victim target become inaccessible via this primary target which will no longer exist. The descendants themselves, if they are somehow still accessible, are accessible through other parents and other locations that don't involve the removed primary victim and the eradicated PCI.
It should be noted that with cases i and ii, it is not held as important that the grandchildren are or are not links. These cases are included for completeness.

Case # i - When the 'alt' key is not pressed, the concern for this case is mostly how to properly determine the cascade of victims, and then from there, narrow it down to target victims. Since the direct grandchild is not a link with this case, not only will it be added to the victim cascade, but it adds additional descendants to the cascade, and the adding continues on, so long as they are not directly under any parents that are links. This process for designating victims happens until a link is encountered, at which point a chain is broken, and we no longer pull nodes onto the victim list, or until a leaf node is found. The victims may not all have 'id' values because not all are necessarily targets. The ones that are targets will potentially cause dangling invalid links if removal is allowed, those are the ones we are ultimately interested in.

To understand better why encountering links would mean we don't continue to include descendants in the list of cascading victims, you can imagine a tree full of nodes, some that are links to other parts of the tree. The links always occur at the boundaries of constructed regions of the tree. It isn't until we reach a link, that we know we've hit a boundary location of some region or branch that is a guest of the main tree. The 'remove' takes place within the region that is being impacted. Once we hit a boundary, we are at that point starting a new region that may or may not be the same region we started from (because cycles are allowed). The key point is that if it is a different region, the impact should only go as far as the link stub. If it is the same region, we know that the expansion process will replicate whatever changes that may have happened to the initial region, in every other instance of that region, regardless of the absolute location within the tree. This should make very clear the reason, when calculating a victim-cascade list, we are concerned only with links and non-links that exist between the primary victim and the deeper edges of the victim's immediate region.

So, the presence of any link that has as it's guest a node from the current vault, will necessarily and reliably indicate that it and it's children point to other original nodes that remain in tact when links are removed. This does not mean that things deeper down in the tree are always unaffected by 'remove' operations of this kind; it simply means we do not need to worry about such cascading results, as the task at hand is simply to predict a list of dangling, invalid links. The redrawn tree, and it's subsequent navigation, will propagate any changes to deeper recursive cycles from a smaller, less-complicated subset.

When the 'alt' key is not pressed the direct grandchildren may be links or non-links, it makes no difference in this discussion, except that, if we find a grandchild link, then, as mentioned above, no other descendants are added to the victim list for descendants of the link. While it may technically be accurate to refer to this link as a 'victim', it is not added to the victim list, as the purpose of that list is not served by adding it. Instead it is added to a list of victim links, so that an engrafted objects list can be updated.

Case # ii - When the 'alt' key is pressed, in this 2a context, and the direct grandchild is not a link, we incorporate that grandchild object as a non-link child directly below the scope item, at the location previously held by the primary victim. If there are other grandchildren that are adopted by the grandparent, they are kept in the same order as they followed as grandchildren.

Case # iii - When the 'alt' key is pressed and the direct grandchild is a link, we incorporate that grandchild as a link, preserving the grandchildren ordering as with case ii.

Case # 2b: The behavior of the 'remove' operation on primary links will be quite simple if the 'alt' key is not pressed, but a bit more involved when it is. When the primary victim is a link, there are four cases to consider:
Case iv - the 'alt' key is not pressed
Case v - the 'alt' key is pressed and the direct grandchild is not a link
Case vi - the 'alt' key is pressed and the direct grandchild is a link

Case # iv - With the 'alt' key not pressed, generally, we just need to remove the link and let the structure of the tree be the result of that very simple operation. The matter of adding things to a victim list is also very simple with a victim target that is a link: since the entire branch being deleted was brought in as a redundant copy of some node by the link during vault construction, it is easy to see that there will remain somewhere else in the tree, an original copy of that node that still exists in some form. Keep in mind that the goal of a 'remove' is not to remove a branch, so much as it is to remove a specific instance of a child from it's parent. The effect on branches is merely the result of reconstructing the branch after the parent-child connection was deleted.

Case # v - When the 'alt' key is down, and if the 'remove' operation is allowed to happen, the children of the primary victim that are not links, will need to be set up as links directly under the scope item. It will not be correct to continue deeper into the descendant subtree making everything a link. The relationships between parent links and children deeper than the grandchild level are unaffected by a single 'ALT+Remove' operation, outside of affects from recursion.

Links are created instead of copies of adopted grandchildren so that the least amount of arbitrary change occurs. Creating duplicates of the grandchildren would introduce a larger amount of modification to the tree that has not been explicitly requested or implied by the 'ALT-remove' operation. This defaults to conserving space on the hard drive and introducing less complexity in the data.

To see what results from an 'ALT-remove' operation in the context of Case # v, compare the before and after diagrams shown in Figures 1 and 2. Prior to the operation we have a tree structure of Figure 1 where L[b] is the selected node, targeted for removal and it is a link node pointing to node 'b'. After the 'ALT-remove', L[b] is

removed and what remains are the items and structure shown in Figure 2, where L[c] is a link to node 'c' and L[d] is link node pointing to node 'd'.

Figure 1

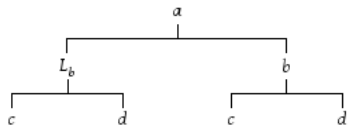
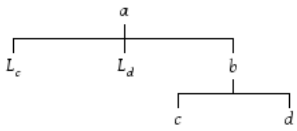


Figure 2



Affects from cyclical recursion are ignored when stitching together branches after removal of a single victim node, except in the case where a grandchild node is also a victim of the 'remove' operation. This is an issue with 'ALT+Remove' and must be caught and handled, so that no victims that are also grandchildren will be grafted into a scope item after 'ALT+Remove'.

Case # vi - When the 'alt' key is down, and if the 'remove' operation is allowed to happen, the children of the primary victim that are also links, will become links directly under the scopeItem, that is they will become links directly under the parent of the victim. This will necessitate a change to any components tracking the location details of links, but not a change to components tracking indigenous target objects.

A general idea to remember when the primary target is a link is that the ownership of the children of the link is some other indigenous node, and this means that moving these things because of 'Alt-remove' does not require an update of the tgtIdTgtMap, only an update of the engraftedObjects entry for the guest grandchildren becoming children under a different parent.

Case # 3a: The case where the child or descendant of a primary victim node is not a link, was considered in the section addressing the situation of case v above

Case # 3b: The case where a child or descendant of a primary victim node is a link, was considered in the section addressing the situation of case vi above

Case # 4a & b: For the purposes of marking a currently selected, primary object a victim, we do not care if there are or are not link objects in the path to instances of the primary victim descendant, since it would not matter that an extrication is happening at a descendent location under a link, or if it is the instance that exists when the tree is entirely collapsed. The user is able to remove a primary object from deep within a recursive hierarchy or path. The code does need to recalculate a valid path and sometimes change the current scope when chunks of that path have been removed.

Determining if a 'remove' operation breaks a link will depend on the 'alt' key, and will generally be independent of whether the current user navigation has created a cycle in the path or not; but perhaps it is worth mentioning, that having a link in certain ancestor locations can correspond with a recursive instance of that link in some descendant position that might keep some node from being automatically added as a victim. This is a proper limitation and is to be expected.

There is a very special case where the scope item is the original pointed to by a primary target that is a link. The 'ALT+remove' should *not* blindly start to load the children of the scope item into itself, thinking they are not already there.

AUXILIARY BOOKKEEPING STRUCTURES MUST BE UPDATED

As was mentioned previously, the client has it's vault data in a tree that is at times expanded and at other times in a collapsed state. As was also said, the guest objects that are swapped in and out in order to collapse and expand the tree, are not copies. They are JavaScript references to a single shared original object. This situation makes it impossible to differentiate between guest object locations and indigenous locations, when looking at the properties of the object itself. The client application uses a set of auxiliary data structures to keep track of all PCI locations associated with guests and non-guest targets. It is critical, of course, to keep the PCI information updated inside the application. Data is at risk otherwise.

In the case where some guest node is either removed or added permanently by the user, the engraftedObjects list and or the tgtIdTgtMap structure must have a corresponding entry removed or added respectively. When an entry is added to either of these collections, the correct PCI information must be provided.

Furthermore, the adding or removing of any node, whether it has an 'id' field or not, can affect the correctness of the child indices inside these harnessing structures. What this simply means is that, for an implementation of this type, code must be written to update those indices, that is, to bump them up or down, according to what has happened during an edit. Equally as important, checks and tests must be written that verify that this has been done correctly.

The consequences of not keeping the target map updated correctly are going to be more benign, but the consequences of not keeping the engrafted objects list updated are that the tree can be saved incorrectly and the opening of a doorway to catastrophic data loss will have occurred, hypothetically speaking. The test strategy below includes both runtime and automated checks on PCI values contained in these structures.

EXACTITUDE AND CORRECTNESS AROUND TREE ITERATION METHODS

Both checking before and after state, and calculating the number and type of victims associated with an operation, are crucial parts of monitoring and implementing data safety. This means knowing what objects are being added and deleted, and it also means knowing which ones are being moved and shifted around, as these aspects are critical for maintaining correct application state and vault information outcomes. It is easy to recognize, then, that for purposes of calculating victims during 'remove', the tabulation routines must reliably and predictably iterate over the native branches of the tree. 'Native branches' are those that have only nodes which have absolute paths that have existed prior to tree expansion.

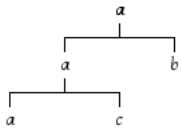
The logic behind iteration is straightforward. Common ways to iterate over a tree are breadth first, and depth first. With these methods of iteration, it is not a difficult thing to visit each node in the tree one and only one time. Where things get complicated is when we need to avoid iteration inside a cycle. When the tree contains a cycle, the iteration algorithm needs to make sure each node is still visited one time, but also avoid missing nodes completely. The bookkeeping that accomplishes this is simple a 'visited' map for all methods that need to iterate over a vault that has been expanded. The map doesn't need to track all nodes that are visited, but just the nodes having an 'id' field, as such nodes exist at the root location of any guest branch.

There are times the client code will walk a collapsed tree, and other times where an expanded tree is iterated over. When we calculate victims, we are iterating an expanded tree. When we do before and after checks of nodes, we are able to use a collapsed tree to tabulate link stubs separately from link targets efficiently. It is when we search an expanded tree that we encounter cycles. It is then we keep track of visited nodes.

In the case where we iterate over an expanded tree, we often do not need to differentiate between nodes that are in link positions and those in non-link positions. By examining the following tree structures, we can answer the question: By ignoring a branch that has been visited, will any nodes be missed entirely? We will want to step through a proof that all nodes will be visited at least once, even if we ignore some nodes we have already seen. This is pretty much a trivial exercise, but it will hopefully make clear that there is a reliable iteration method available to us.

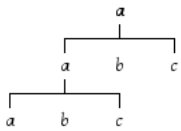
Assume that node 'a' has a child 'a'. This sets up a redundant cycle. The problem is we don't want to revisit 'a', but we do want to reach all of it's children. See Fig 1.

Figure 3



The misrepresentation of the tree in Figure 3 above shows that we might not visit node 'c' if we avoid the cycle by refusing to descend into 'a' again. There is an omission here in Figure 3 that is sort of a key to understanding the actual situation. What has been omitted here is that if the child 'a' of parent 'a' is an exact copy (or more precisely an instance of 'a' reoccurring at a nested location), then 'c' will not only be buried in the lowest leaf level as shown, but it will also exist as a direct child in the uppermost level. Figure 4 shows the complete picture. This demonstrates that we can descend once into 'a', not descend a second time, but encounter each and every node in the tree.

Figure 4



CHECKING RETURN VALUES AND EXPECTED INPUTS

Knowing why a method returned and what its input and output are, is basic to all well-written code. Chiefly, what it provides for the client application during edit operations is more reliability in terms of being able to monitor for things that have gone wrong and putting on the brakes if appropriate, for example, blocking the save operation to protect data.

A survey of all methods for which data loss dictates a high priority for handling, close monitoring and reporting of errors is going to be key. These will be any methods found or invoked during the execution of the critical regions described above.

Information extracted from all of the critical region's methods are intentionally reported in the 1) debug console 2) the log file 3) default runtime tests 4) feedback loops that block the save operation

OTHER MORE REMOTE POSSIBILITIES FOR DATA LOSS

Is there any possibility that a 'remove' is directed to some node that is not actually the selected node? Realistically, there is not a way that the client application, as written, will execute actions that have not been requested by users. Such a behavior would not be a function of code complexity or programmer error. The most likely way something like this could happen would be if malicious code were to be injected into the downloaded code.

Recommendations for protecting user data, from this standpoint are the 'additional strategies' mentioned above. Those recommendations are:

a) work behind a firewall
b) use MD5 to verify any downloaded version of the application (currently not available).
c) have updated virus protection running on client systems

THE IMPORTANCE OF TESTING

Getting the correct behavior in all of the client application code is not the only strategy that we must use. It is, however, the first step to take. Other important steps also include automated and runtime testing, as well as the other safeguards mentioned above to make sure users will not have any surprises when it comes to their data.

A TEST STRATEGY TO ENSURE DATA SAFETY

For each type of edit operation, test code must check the tree, the tgtIdTgtMap and the engraftedObjects entries, the child Ids, parentIds and travelMaps

For each specific edit, a test must look at:

1) integrity of the vault
2) integrity of the harness - especially tgtIdTgtMap and engraftedObjects
3) correctness of the engraftedObjects at the scope level before and after the 'remove' or 'Alt-remove'
4) absence of the engraftedObjects at the grandchild level
   - PCI with excised as child and scope item as parent for removed child index should not be present
   - PCI with excised as parent and promoted grandchildren as child
5) the correctness of the tgtIdTgtMap - considering the before and after circumstances of primary victim and it's children

The 'remove' operation presents the greatest challenge, because of the its complexity. There are 8 basic cases to consider.

when	case number
the target is not a link	
a grandchild is not a link	
not Alt-remove	2a(i)
Alt-remove	2a(ii)
a grandchild is link	
not Alt-remove	2a(i)
Alt-remove	2a(iii)

the target is a link
    a grandchild is not a link
        not Alt-remove 2b(iv)
        Alt-remove 2b(v)
    a grandchild is link
        not Alt-remove 2b(iv)
        Alt-remove 2b(vi)

LOGGING AND CONSOLE OUTPUT

Anyone who is even a little bit serious about system validation should seek to see the runtime behavior of their application, which means they should have both console output and real-time updates to a log file that will enable all who are responsible for testing and monitoring to see what happens in test environments as well as real world situations.

The Unitraverse client application provides this output, and also offers a 'high alert' mode, which more vigorous extraction of runtime information, writes any errors or anomalies to the log file. Vault state tests and extra error checking tests come with the 'high_alert' mode. This is in addition to any critical information that will always get logged.

TEST MATRIX FOR THE UNITRAVERSE CLIENT APPLICATION

The following checklist represents a test matrix and other action points that need to be followed for assuring data safety in and around designing, implementing and using the client application

[ ] advise of version that covers all the bases
[ ] caution developers against direct edit of children or childLabels

Vault integrity tests
[ ] all context nodes with an 'id' field have a corresponding tgtIdTgtMap entry with matching childIndex and parent or a corresponding engraftedObjects entry
[ ] all context nodes with a targetId field have a corresponding tgtIdTgtMap entry with matching childIndex and parent and an engraftedObjects entry
[ ] all engraftedObjects entries with a stub that is not null, and having a targetId, should also have a corresponding tgtIdTgtMap with matching parent and childIndex
    [x] 4) there are not duplicate 'ids' for context nodes in a collapsed data tree
    [x] 5) that there are not targetIds that have no matching context nodes with the same value as the 'id' - (so search the tree, not just the map)

blockingSaveOp
[x] doAddContext
[x] addFromJSON
[x] paste
[x] remove
[x] addVaultItem_applet
[x] replaceVaultItemAtLocation_applet [not critical but having possible extra benefit]

test functions must be tested against simple known conditions (as per oneTimeTestOfTests_test)
[x] hasDuplicateContextNodeIds_test
[x] hasUnclaimedTargetStubs_test
[x] hasUnusedEngraftedObjectEntries_test
[x] countTargetIds_test
[x] validateChildIds_test
[x] validateParentIds_test
[x] isValidIdsArray
[x] validateTgtIdTgtMapKeys_test
[x] validateTgtIdTgtMapEntries_test
[x] validateTravelMap_test
[x] alternateValidateTravelMap_test
[x] validateTravelMap_testRegion
[x] does exist in tree
[x] checkHarness_test

Add Cases - Automated testing (both platform and public)

[Currently the 'add-item' cases in the client app are tested with less rigorous automated tests, but they still are tested nevertheless]

Remove Cases - Automated Testing (both platform and public)
In the process we want to also change things up:
    [x] add test that removes initial child
    [x] augment test that removes middle child
    [x] add test that removes last child

target is not a link
    grandchild is not a link
        not Alt-remove 2a(i)
            [ ] add test that causes cascade that ends with links and leaf nodes
            [x] check that primary victim does not exist in the tree
            [x] check that primary victim does not exist in engraftedObjects
            [x] check that primary victim id does not exist in tgtIdTgtMap
            [ ] check that primary victim does not exist in travelMap
            [ ] check that primary victim id does not exist in pathIdsStack
            [x] check that no link victim stubs remain in the collapsed tree
            [x] check that no link victim stubs remain in engraftedObjects
            [x] check that no link victim guests remain in expanded tree
            [x] check that no link victim guests remain in engraftedObjects
            [x] check that no indigenous target victims remain in tree
            [x] check that no indigenous target victim ids remain in tgtIdTgtMap
            [x] check that no indigenous target victims remain in travelMap
            [x] check that no indigenous target victim ids remain in pathIdsStack
            [x] check that no non-link-non-target victims remain in the tree
            [x] check the number of victims, compare vault size: (num_after - num_before) == num_victims
            [x] check the objects of all scope items children before and after
            [x] check the positions of scope item target children in tgtIdTgtMap

[x] check the positions of scope item link children in engraftedObjects
[x] check that no grandchildren were retained
[x] check that no links were added to engraftedObjects
[x] check that no links other than victims were removed from engraftedObjects
[x] check that no targets were added to tgtIdTgtMap
[x] check that no targets other than victims were removed from tgtIdTgtMap
[x] check that no other parts of the collapsed tree, other than intended victims were removed
[x] check that no other parts of the expanded tree was removed other than intended victims
[x] check that there are no duplicates in the bank array
[x] check no error messages sent
[x] check blockingSaveOp not set

Alt-remove 2a(ii)

[x] check that primary victim does not exist in the tree
[x] check that primary victim does not exist in engraftedObjects
[x] check that primary victim id does not exist in tgtIdTgtMap
[ ] check that primary victim does not exist in travelMap
[ ] check that primary victim id does not exist in pathIdsStack
[x] check that no link victim stubs remain in the collapsed tree
[x] check that no link victim stubs remain in engraftedObjects
[x] check that no link victim guests remain in expanded tree
[x] check that no link victim guests remain in engraftedObjects
[x] check that no indigenous target victims remain in tree
[x] check that no indigenous target victim ids remain in tgtIdTgtMap
[x] check that no indigenous target victims remain in travelMap
[x] check that no indigenous target victim ids remain in pathIdsStack
[x] check that no non-link-non-target victims remain in the tree
[x] check the number of victims, compare vault size: (num_after - num_before) == num_victims
[x] check the objects of all scope items children before and after
[x] check the positions of scope item target children in tgtIdTgtMap
[x] check the positions of scope item link children in engraftedObjects
[-] check that no grandchildren were retained
[x] check that no links were added to engraftedObjects
[x] check that no links other than victims were removed from engraftedObjects
[x] check that no targets were added to tgtIdTgtMap
[x] check that no targets other than victims were removed from tgtIdTgtMap
[x] check that no other parts of the collapsed tree, other than intended victims were removed
[x] check that no other parts of the expanded tree was removed other than intended victims
[x] check that there are no duplicates in the bank array
[x] check no error messages sent
[x] check blockingSaveOp not set

grandchild is link

not Alt-remove 2a(i)

[ ] add test that causes cascade that ends with links and leaf nodes
[x] check that primary victim does not exist in the tree
[x] check that primary victim does not exist in engraftedObjects
[x] check that primary victim id does not exist in tgtIdTgtMap
[ ] check that primary victim does not exist in travelMap
[ ] check that primary victim id does not exist in pathIdsStack
[x] check that no link victim stubs remain in the collapsed tree
[x] check that no link victim stubs remain in engraftedObjects
[x] check that no link victim guests remain in expanded tree
[x] check that no link victim guests remain in engraftedObjects
[x] check that no indigenous target victims remain in tree
[x] check that no indigenous target victim ids remain in tgtIdTgtMap
[-] check that no indigenous target victims remain in travelMap
[-] check that no indigenous target victim ids remain in pathIdsStack
[x] check that no non-link-non-target victims remain in the tree
[x] check the number of victims, compare vault size: (num_after - num_before) == num_victims
[x] check the objects of all scope items children before and after
[x] check the positions of scope item target children in tgtIdTgtMap
[x] check the positions of scope item link children in engraftedObjects
[x] check that no grandchildren were retained
[x] check that no links were added to engraftedObjects
[x] check that no links other than victims were removed from engraftedObjects
[x] check that no targets were added to tgtIdTgtMap
[x] check that no targets other than victims were removed from tgtIdTgtMap
[x] check that no other parts of the collapsed tree, other than intended victims were removed
[x] check that no other parts of the expanded tree was removed other than intended victims
[x] check that there are no duplicates in the bank array
[x] check no error messages sent
[x] check blockingSaveOp not set

Alt-remove 2a(iii)

[ ] add test that causes cascade that ends with links and leaf nodes
[x] check that primary victim does not exist in the tree
[x] check that primary victim does not exist in engraftedObjects
[x] check that primary victim id does not exist in tgtIdTgtMap
[ ] check that primary victim does not exist in travelMap
[ ] check that primary victim id does not exist in pathIdsStack
[x] check that no link victim stubs remain in the collapsed tree
[x] check that no link victim stubs remain in engraftedObjects
[x] check that no link victim guests remain in expanded tree
[x] check that no link victim guests remain in engraftedObjects
[x] check that no indigenous target victims remain in tree
[x] check that no indigenous target victim ids remain in tgtIdTgtMap
[-] check that no indigenous target victims remain in travelMap
[-] check that no indigenous target victim ids remain in pathIdsStack
[x] check that no non-link-non-target victims remain in the tree
[x] check the number of victims, compare vault size: (num_after - num_before) == num_victims
[x] check the objects of all scope items children before and after
[x] check the positions of scope item target children in tgtIdTgtMap

[x] check the positions of scope item link children in engraftedObjects
[x] check that all grandchildren were retained
[x] check that no links were added to engraftedObjects
[x] check that no links other than victims were removed from engraftedObjects
[x] check that no targets were added to tgtIdTgtMap
[x] check that no targets other than victims were removed from tgtIdTgtMap
[x] check that no other parts of the collapsed tree, other than intended victims were removed
[x] check that no other parts of the expanded tree was removed other than intended victims
[x] check that there are no duplicates in the bank array
[x] check no error messages sent
[x] check blockingSaveOp not set

target is link

grandchild is not a link

not Alt-remove 2b(iv)

[ ] add test that causes cascade that ends with links and leaf nodes
[x] check that primary victim does not exist in the tree
[x] check that primary victim does not exist in engraftedObjects
[x] check that primary victim id does not exist in tgtIdTgtMap
[ ] check that primary victim does not exist in travelMap
[ ] check that primary victim id does not exist in pathIdsStack
[x] check that no link victim stubs remain in the collapsed tree
[x] check that no link victim stubs remain in engraftedObjects
[x] check that no link victim guests remain in expanded tree
[x] check that no link victim guests remain in engraftedObjects
[x] check that no indigenous target victims remain in tree
[x] check that no indigenous target victim ids remain in tgtIdTgtMap
[-] check that no indigenous target victims remain in travelMap
[-] check that no indigenous target victim ids remain in pathIdsStack
[x] check that no non-link-non-target victims remain in the tree
[x] check the number of victims, compare vault size: (num_after - num_before) == num_victims
[x] check the objects of all scope items children before and after
[x] check the positions of scope item target children in tgtIdTgtMap
[x] check the positions of scope item link children in engraftedObjects
[x] check that no grandchildren were retained
[x] check that no links were added to engraftedObjects
[x] check that no links other than victims were removed from engraftedObjects
[x] check that no targets were added to tgtIdTgtMap
[x] check that no targets other than victims were removed from tgtIdTgtMap
[x] check that no other parts of the collapsed tree, other than intended victims were removed
[x] check that no other parts of the expanded tree was removed other than intended victims
[x] check that there are no duplicates in the bank array
[x] check no error messages sent
[x] check blockingSaveOp not set

Alt-remove 2b(v)

[ ] add test that causes cascade that ends with links and leaf nodes
[x] check that primary victim does not exist in the tree
[x] check that primary victim does not exist in engraftedObjects
[x] check that primary victim id does not exist in tgtIdTgtMap
[ ] check that primary victim does not exist in travelMap
[ ] check that primary victim id does not exist in pathIdsStack
[x] check that no link victim stubs remain in the collapsed tree
[x] check that no link victim stubs remain in engraftedObjects
[x] check that no link victim guests remain in expanded tree
[x] check that no link victim guests remain in engraftedObjects
[x] check that no indigenous target victims remain in tree
[x] check that no indigenous target victim ids remain in tgtIdTgtMap
[-] check that no indigenous target victims remain in travelMap
[-] check that no indigenous target victim ids remain in pathIdsStack
[x] check that no non-link-non-target victims remain in the tree
[x] check the number of victims, compare vault size: (num_after - num_before) == num_victims
[x] check the objects of all scope items children before and after
[x] check the positions of scope item target children in tgtIdTgtMap
[x] check the positions of scope item link children in engraftedObjects
[x] check that all grandchildren were retained
[x] check that no links were added to engraftedObjects
[x] check that no links other than victims were removed from engraftedObjects
[x] check that no targets were added to tgtIdTgtMap
[x] check that no targets other than victims were removed from tgtIdTgtMap
[x] check that no other parts of the collapsed tree, other than intended victims were removed
[x] check that no other parts of the expanded tree was removed other than intended victims
[x] check that there are no duplicates in the bank array
[x] check no error messages sent
[x] check blockingSaveOp not set

grandchild is link

not Alt-remove 2b(iv)

[ ] add test that causes cascade that ends with links and leaf nodes
[x] check that primary victim does not exist in the tree
[x] check that primary victim does not exist in engraftedObjects
[x] check that primary victim id does not exist in tgtIdTgtMap
[ ] check that primary victim does not exist in travelMap
[ ] check that primary victim id does not exist in pathIdsStack
[x] check that no link victim stubs remain in the collapsed tree
[x] check that no link victim stubs remain in engraftedObjects
[x] check that no link victim guests remain in expanded tree
[x] check that no link victim guests remain in engraftedObjects
[x] check that no indigenous target victims remain in tree
[x] check that no indigenous target victim ids remain in tgtIdTgtMap
[-] check that no indigenous target victims remain in travelMap
[-] check that no indigenous target victim ids remain in pathIdsStack
[x] check that no non-link-non-target victims remain in the tree

[x] check the number of victims, compare vault size: (num_after - num_before) == num_victims
[x] check the objects of all scope items children before and after
[x] check the positions of scope item target children in tgtIdTgtMap
[x] check the positions of scope item link children in engraftedObjects
[x] check that no grandchildren were retained
[x] check that no links were added to engraftedObjects
[x] check that no links other than victims were removed from engraftedObjects
[x] check that no targets were added to tgtIdTgtMap
[x] check that no targets other than victims were removed from tgtIdTgtMap
[x] check that no other parts of the collapsed tree, other than intended victims were removed
[x] check that no other parts of the expanded tree was removed other than intended victims
[x] check that there are no duplicates in the bank array
[x] check no error messages sent
[x] check blockingSaveOp not set

Alt-remove 2b(vi)

[ ] add test that causes cascade that ends with links and leaf nodes
[x] check that primary victim does not exist in the tree
[x] check that primary victim does not exist in engraftedObjects
[x] check that primary victim id does not exist in tgtIdTgtMap
[ ] check that primary victim does not exist in travelMap
[ ] check that primary victim id does not exist in pathIdsStack
[x] check that no link victim stubs remain in the collapsed tree
[x] check that no link victim stubs remain in engraftedObjects
[x] check that no link victim guests remain in expanded tree
[x] check that no link victim guests remain in engraftedObjects
[x] check that no indigenous target victims remain in tree
[x] check that no indigenous target victim ids remain in tgtIdTgtMap
[-] check that no indigenous target victims remain in travelMap
[-] check that no indigenous target victim ids remain in pathIdsStack
[x] check that no non-link-non-target victims remain in the tree
[x] check the number of victims, compare vault size: (num_after - num_before) == num_victims
[x] check the objects of all scope items children before and after
[x] check the positions of scope item target children in tgtIdTgtMap
[x] check the positions of scope item link children in engraftedObjects
[x] check that all grandchildren were retained
[x] check that no links were added to engraftedObjects
[x] check that no links other than victims were removed from engraftedObjects
[x] check that no targets were added to tgtIdTgtMap
[x] check that no targets other than victims were removed from tgtIdTgtMap
[x] check that no other parts of the collapsed tree, other than intended victims were removed
[x] check that no other parts of the expanded tree was removed other than intended victims
[x] check that there are no duplicates in the bank array
[x] check no error messages sent
[x] check blockingSaveOp not set