Stateful Accessibility for Views (June 2024)

Author: Benjamin Beaudry (benjamin.beaudry@microsoft.com)
Last modified: 6/14/2024

The April 2024 and December 2023 version of this document are available below, but do not reflect the latest changes to the project.

Only fools never change their minds. We thought of yet another better approach for supporting accessibility for views while limiting the performance impact to a minimum. This document goes over the new approach. The target audience is Chromium Views authors, owners, and contributors to the ViewsAX project.

Before we get started, let's recap what this project is and why we're working on it.

What is it?

The ViewsAX project is a massive refactor of the accessibility framework used to make Chromium Views accessible. It aims to build a cache for all accessibility attributes of each Chromium View to allow for fast retrieval instead of computing all attributes, all the time, like we do today. Today's accessibility framework for Views is designed to be a "pull" system, where all the accessible data is computed on the fly whenever an assistive technology needs to know something. With our work, it will change to a "push" system, where Views are responsible for sending live updates to the cached accessible data.

Views have three main ways of setting accessible attributes:

1. Through ViewAccessibility::Override[Attribute] **(this one has been deprecated for a few months, and we finally removed it from the codebase)**
2. Through ViewAccessibility::SetAccessible[Attribute] **(also deprecated)**
3. Through overrides of View::GetAccessibleNodeData

These three different interfaces all store the data in different objects, and later combine them following a precarious set of rules. With the ViewsAX project, we'll have a single cache object that stores all accessible data for a specific view and once initialized through an initializer virtual function in View, the only way to modify the cache will be through the newly introduced ViewAccessibility::Set[Attribute] setters.

Why do we do this?

Immediate benefits

1. Improved performance: This project builds the system to cache and maintain the cache with up-to-date information, allowing us to transform what is currently a O(n*m) cost (where n = number of attributes queried by the assistive technology, and m = number of attributes set on the view) to a O(1) one.
2. Better accessibility: The cache will allow us to detect automatically when an attribute changes and fire the right accessibility events when it does. This will improve the accessible experience by automatically doing what Views authors should already be doing and make it easier for them to make accessible Views. We plan to achieve this by leveraging the AXTree logic we already have for web content.
3. Increased code quality:
    1. Simplify the way Views authors can set accessible data: replace View::SetAccessible[Attribute], ViewAccessibility::Override[Attribute], View::GetAccessibleNodeData, with an accessibility cache that authors can directly change by calling GetViewAccessibility().Set[Attribute]. For attributes that are expensive to compute and/or store (like a long string, for example), there will also be a secondary system in place that allows Views authors to only initialize those attributes when accessibility is in use.
    2. No more hacks needed to make UIA work with Views. Having a cache will allow us to have an AXTree of AXNodes, which will give us access to the AXPosition code and allow the AXPlatformNodeTextRangeProviderWin to work just like it does for web content.
    3. No more code duplications are necessary between BrowserAccessibility and AXPlatformNode.

Future benefits

1. Better memory management - Allow us to turn "off" the expensive accessibility-related logic for users who do not need accessibility support (the majority of our users) and lazily compute it once accessibility support is required (e.g., when an access technology starts fetching information). Cache when it's not needed and turn it back "on" when needed.
2. Get us one step closer to being able to move the core accessibility logic to a utility process. Having a cache of that accessible data will allow us to

serialize/unserialize it through mojom to a different process. This will require extra work, but that extra work will be possible now that we have cache.

The vision

Here's a point-by-point breakdown of how accessibility for Views will work with this project:

1. The accessible data for a view now lives in a cache.
2. The cache is a private AXNodeData member in ViewAccessibility.
3. By default, the cache is empty.
4. Views authors can set attributes to the cache by calling ViewAccessibility::Set[Attribute].
5. Some attributes are expensive to compute or use a lot of memory space: these attributes should be lazily computed.
6. To lazily compute expensive attributes, Views subclasses should override View::OnAccessibilityEnabled to set the expensive attributes. This allows for improved performance, but is not strictly required for the cache to function.
7. This function is called when accessibility is enabled by an access technology querying an accessible attribute via an accessibility API for the first time, or when it's added to the Views tree (through View::AddChildView) if accessibility is already enabled.
6. A view must always update the accessibility cache whenever an attribute's value changes.
7. The cache is updated through the ViewAccessibility setters' interface. For example, to update the name, a Views author would call `view->GetViewAccessibility().SetName(new_name)`.
8. When accessible data is queried (e.g., by an assistive technology) on a specific view, we return the data from the cache.

How this diverges from what we previously discussed

The April 2024 version of this document was based on the principle of never storing any accessibility attributes when it wasn't necessary. To achieve this, we decided to leverage the existing View::GetAccessibleNodeData to compute the initial accessibility state whenever accessibility would become enabled. However, this implied that all Views would need to have an override of this function, and therefore that all views that differed only from their initial accessibility attributes would have to become their own subclasses. For example, creating an instance of View and setting the role to list would

no longer be valid; we would have to create a subclass ListView, and override the GetAccessibleNodeData function in it to expose the list role.

Previously, we were not planning on getting rid of the GetAccessibleNodeData function, but with this new approach, we can reconsider this . GetAccessibleNodeData is no longer necessary: all inexpensive attributes, such as the role, are allowed to be set directly in the cache, and we will also empower views authors to set expensive attributes, such as a long URL string, in the OnAccessibilityEnabled, which will be called as soon as the following are true: accessibility is enabled, and the view is connected to the tree.

For attributes that change, it will be required to use the setters.

While this approach no longer considers the ability of fully disabling the accessibility cache when not needed as part of the ViewsAX project, it gives more power to Views authors to lazily load expensive attributes only when accessibility is needed. It also simplifies the interface to make Views accessible.

The plan to realize the vision

Owners can expect to see many CLs coming from us to get us into this desired final state. All changes should fit within one of the following buckets:

1. Modify individual Views so they update the cache when the value of some attribute's changes.
    a. For each accessibility attribute used in Views, we plan to:
        i. Add an accessibility cache setter.
        ii. Update each view that uses the attribute so they update the cache when the value of the attribute changes.
        iii. Remove the attribute from GetAccessibleNodeData, which won't be needed anymore.
2. Update/add tests, to validate the accessibility state is set correctly and updated when needed.
3. Get rid of GetAccessibleNodeData. Move the remaining attributes exclusively set in GetAccessibleNodeData in OnAccessibilityEnabled.
4. [Future] Connect the cache in ViewAccessibility with the AXTreeManager. The end goal of this project is to reap the benefits of the advanced accessibility tree handling we built for web content, BrowserAccessibilityManager, which includes automatic event generation and the right connections to the platform accessibility APIs. We'll be able to have an AXTree of AXNodes, where each AXNode node maps to a ViewAccessibility node. Jacques is currently refactoring this code to

bring it out of the content layer and into the ui/accessibility layer so we can reuse it for Views. You can learn more about this project here and here.

What we ask from Views owners

Given how the paradigm for accessibility in Views is different now – simpler – we expect Views authors to more easily set accessibility attributes. However, we'll need the help of Views owners to enforce that expensive accessibility attributes, such as long strings or attributes that involve computationally heavy operations (e.g., text offsets), are lazily loaded in the View::OnAccessibilityEnabled virtual function. Owners will also have to enforce that we don't unnecessarily duplicate accessible events generation, since the cache setters will be able to fire events automatically when needed.

We'd also like to hear your thoughts on the best way to communicate these changes with the community of developers regularly modifying Views.

Performance impact of the cached data

We were wondering how impactful the cache would be on the performance and memory of the browser, so we put together this quick change that simply traverses the widget's Views hierarchy and computes the accessible state for every view. As it turns out, the cache should have a minimal impact on the performance.

On a simple browser window with one single tab, we had 488 Views. The total memory usage by the accessibility cache was 30.6 KB with no attributes lazily loaded. Once we build the lazy loading mechanism, we expect this number to go down even more for users who do not need accessibility support. Calling the OnAccessibilityEnabled function to lazy load the attributes (while recomputing attributes that were already in the cache) took 1.288 ms. We don't expect this lazy loading to have a noticeable impact on core performance metrics.

Stateful Accessibility for Views (April 2024)

Author: Benjamin Beaudry (benjamin.beaudry@microsoft.com)
Last modified: 4/17/2024

The December 2023 version of this document is available below.

This document is a mid-way update on the ViewsAX project. It goes over the progress we have made, where we plan to end up, and what changes we expect to make to get there. The target audience is Chromium Views authors, owners, and contributors to the ViewsAX project.

Before we get started, let's recap what this project is and why we're working on it.

What is it?

The ViewsAX project is essentially a massive refactor of the accessibility framework used to make Chromium Views accessible. It aims to build a cache for all accessibility attributes of each Chromium View to allow for fast retrieval instead of computing all attributes, all the time, like we do today. Today's accessibility framework for Views is designed to be a "pull" system, where all the accessible data is computed on the fly whenever an assistive technology needs to know something. With our work, it will change to a "push" system, where Views are responsible for sending live updates to the cached accessible data.

Views have three main ways of setting accessible attributes:

1. Through ViewAccessibility::Override[Attribute] **(this one has been deprecated for a few months, and we finally removed it from the codebase)**
2. Through ViewAccessibility::SetAccessible[Attribute] (we're currently working on removing this option, as well)
3. Through overrides of View::GetAccessibleNodeData

These three different interfaces all store the data in different objects, and later combine them following a precarious set of rules. With the ViewsAX project, we'll have a single cache object that stores all accessible data for a specific view and once initialized through an initializer virtual function in View, the only way to modify the cache will be through the newly introduced ViewAccessibility::Set[Attribute] setters.

Why do we do this?

Immediate benefits

1. Improved performance: This project builds the system to cache and maintain the cache up to date, allowing us to transform what is currently a O(n*m) cost (where n = number of attributes queried by the assistive technology, and m = number of attributes set on the view) to a O(p*m) one (where p = number of Views and m = number of attributes set on the view) for the **first query**, and O(1) problem (a cache hit) for all subsequent queries.
2. Better accessibility: The cache will allow us to detect automatically when an attribute changes and fire the right accessibility events when it does. This will improve the accessible experience by automatically doing what Views authors should already be doing and make it easier for them to make accessible Views. We plan to achieve this by leveraging the AXTree logic we already have for web content.
3. Increased code quality:
    1. No more hacks needed to make UIA work with Views. Having a cache will allow us to have an AXTree of AXNodes, which will give us access to the AXPosition code and allow the AXPlatformNodeTextRangeProviderWin to work just like it does for web content.
    2. No more code duplications are necessary between BrowserAccessibility and AXPlatformNode.
    3. Simplify the way Views authors can set accessible data: replace View::SetAccessible[Attribute], ViewAccessibility::Override[Attribute], View::GetAccessibleNodeData, by a cache system that gets initialized once (through View::GetAccessibleNodeData, potentially renamed to View::ComputeAccessibility or something else) and updated only through setters in ViewAccessibility.

Future benefits

1. Better memory management - Allow us to turn "off" the accessibility-related logic and cache when it's not needed and turn it back "on" when needed. This will involve working with assistive technology partners to ensure Chromium does not regress any product when doing so.
2. Get us one step closer to being able to move the core accessibility logic to a utility process. Having a cache of that accessible data will allow us to

serialize/unserialize it through mojom to a different process. This will require extra work, but that extra work will be possible now that we have cache.

The vision

Here's a point-by-point breakdown of how accessibility for Views will work with this project:

1. The accessible data for a view now lives in a cache.
2. The cache is a private AXNodeData member in ViewAccessibility.
3. By default, the cache is not initialized.
4. The cache for all Views within the view hierarchy of a widget gets initialized when an assistive technology queries an accessible attribute via an accessibility API for the first time.
5. The cache gets initialized by calling the new function, ViewAccessibility::Initialize, which calls View::GetAccessibleNodeData **once** on each view in the widget view hierarchy. (note: since the purpose of that function is a bit different than it was before, we consider renaming this function to View::ComputeAccessibility. Thoughts?). This function must return the current accessible state of a view, not the one at the time of construction of the view.
6. Once a cache is initialized, a view can and should update it directly as soon as an attribute changes.
7. The cache is updated through the ViewAccessibility setters' interface. For example, to update the name, a Views author would call `view->GetViewAccessibility().SetName(new_name)`.
8. When accessible data is queried (e.g., by an assistive technology) on a specific view, we return the data from the cache.
9. If a view calls a ViewAccessibility cache setter when
    1. the cache is initialized, it updates the cache directly (and fires the events needed).
    2. the cache is off, it's a no op.
        1. If a view needs to do an expensive computation before setting it in the cache, it can be avoided by looking at the ViewAccessibility::is_initialized() public member first.

How this diverges from what we previously discussed

What we discussed in the previous version of this document is similar but wasn't as complete. One important difference is that we were previously planning to completely

get rid of View::GetAccessibleNodeData and move all the attributes that were set in there to the constructor of the view. This was a shortcoming of the previous design; we didn't think through the steps that would be needed to restore the accessibility state at any point if we had an ON/OFF switch for the cache.

To be able to restore the cache at any moment, we need to be able to get the current accessible state for a view. If the cache can get initialized only from the view's constructor, there's no way to recompute those accessible attributes without recreating the whole view.

Instead, the solution we prefer that would allow us to add an ON/OFF switch for the cache is a function similar to View::GetAccessibleNodeData (preferably re-named), that would only be called once: when we need to initialize the cache.

The plan to realize the vision

Owners can expect to see many CLs coming from us to get us into this desired final state. All changes should fit within one of the following buckets:

1. Get rid of View::SetAccessible[Attribute] functions.
2. Move the SetAccessible[Attribute] functions out of the constructor for each view, to the start of the View::GetAccessibleNodeData override.
3. Build the new cache initialization system. This involves:
    a. Building the mechanism that triggers the cache initialization for the entire widget's view hierarchy.
    b. Building the mechanism that initializes the cache for each new view added to the view hierarchy after accessibility has been initialized for the widget.
    c. Modify the ViewAccessibility setters to return early if the cache is not yet initialized.
4. Modify individual Views so they update the cache when the value of some attribute's changes.
5. Update/add tests, to validate the accessibility state when the cache is initialized, validate it gets updated correctly once it's initialized, and validate that GetAccessibleNodeData always return the latest data (which will confirm we can restore the cache at any moment).
6. [Future] Connect the cache in ViewAccessibility with the AXTreeManager. The end goal of this project is to reap the benefits of the advanced accessibility tree handling we built for web content, BrowserAccessibilityManager, which includes automatic event generation and

the right connections to the platform accessibility APIs. We'll be able to have an AXTree of AXNodes, where each AXNode node maps to a ViewAccessibility node. Jacques is currently refactoring this code to bring it out of the content layer and into the ui/accessibility layer so we can reuse it for Views. You can learn more about this project here and here.

What we ask from Views owners

Given how the paradigm for accessibility in Views is different now – we need to be able to compute it at any moment and update the state when something changes – we'll need the help of Views owners to enforce both requirements when authors make changes that could affect accessibility.

Take, for example, the following file that Javier modified to demonstrate what a good accessibility change would look like: editable_combobox.cc. In this case, we changed

- the value in the cache
- we **don't** fire an event (since the setter will take care of it ~~for free~~ for free!)
- and we make sure that there's a valid current value that we can get by calling GetAccessibleNodeData.

We'd also like to hear your thoughts on the best way to communicate these changes with the community of developers regularly modifying Views.

Performance impact of the cached data

We were wondering how impactful the cache would be on the performance and memory of the browser, so we put together this quick change that simply traverses the widget's Views hierarchy and computes the accessible state for every view. As it turns out, the cache should have a minimal impact on the performance.

On a simple browser window with one single tab, we had 488 Views. To initialize the accessible state for all of them, it took us exactly 1.288 milliseconds (on a high-end device) and consumed an extra 30.6 KB. Initializing the cache is a **one-time** cost, and the following updates to the cache are trivial. Please note that this cost does not include the serialization and unserialization - we will acheive this when we have an AXTree.

To limit this performance impact to only the users who need to have the accessibility support, we plan on having the cache disabled by default on browser launch (i.e. It won't impact the launch time) and enable it when an assistive technology performs a first query in the browser.

Stateful Accessibility for Views (December 2023)

Author: Javier Contreras (javiercon@microsoft.com)
Last updated on: 12/5/2023

Problem

This project is a sub-project and prerequisite to complete the overarching ViewsAX project, which intends for Views to be organized into an AXTree, effectively reusing and adapting the accessibility architecture used for web content.

- ViewsAX design doc
- ViewsAX: Project C addendum

Independently, this project seeks to replace the pull system used to get a View's accessibility properties by a push one. Currently, this data is computed on the fly and as thus there is no accessible state for a particular view – it is just computed every time this is needed. Through this project, we aim to phase this out in favor of a new "stateful" accessibility system for every View.

Proposed Solution

Summary

Our goal is to replace the existing method of exposing accessible properties for Views. Currently, we make use of a GetAccessibleNodeData override, or ViewAccessibility::Override* functions to expose the appropriate accessibility properties for a particular View.

Our proposal is to instead cache accessible properties on ViewAccessibility, and have these be updated by calling ViewAccessibility::Set* functions that we will introduce (some have already been introduced). These properties will keep the ViewAccessibility's AXNodeData up-to-date and fire accessibility events as appropriate so that individual Views don't have to. Views would now set the accessible properties via the ViewAccessibility::Set* setters as soon as the value for that property is known and whenever it changes, instead of computing over and over each time it is queried by an assistive technology.

Why

There are multiple advantages and motivations for adopting this proposed change.

Firstly and most importantly, this is a required change for the ViewsAX project – which consists of organizing Views into an AXTree – to be completed.

Some of the benefits that the ViewsAX project will provide are:

1. We will be able to reuse our AXTree and AXNode logic, allowing for automatic accessible events generation – essentially making all of Views more accessible by default.
2. We won't need to recompute the same accessible data multiple times since it will be cached
3. It will be required one day when we start the project to move accessibility out of the browser process and into its own utility process, so it will prevent this issue from becoming a blocker.

Beyond these, we also get multiple benefits stemming directly from this specific sub-project:

1. Currently, ViewAXPlatformNodeDelegate::GetData does arguably much more work than it should, and it is also called a lot. When it gets called, first we start out with an empty AXNodeData which is populated by calling ViewAccessibility::GetAccessibleNodeData which calls the function of the same name on the View. This then may do different calculations to determine things such as the accessible name, the position in set, and many others. After, GetData ascends the View's ancestors with IsViewUnfocusableDescendantOfFocusableAncestor. All of this is done for every single property that is needed, which is unnecessary and detrimental to performance.

2. Responsibility for exposing values and firing the appropriate events would no longer fall on the View and its authors, instead shifting to View::Set*. The View and its authors would just need to call the appropriate View::Set* to set the properties and update them if need be.
    a. Currently, the flow for a Views author might look something like:
        i. `view->GetViewAccessibility().OverrideName(name);`
        ii. `view->NotifyAccessibilityEvent(kTextChanged, true);`
    b. But with this proposed change, it would look something like:
        i. `view->SetName(name);`
    c. The already existing SetName demonstrates how this would be done.

3. Callers can query for particular properties rather than reconstructing the whole data every single time.

4. We'll be able to get rid of ViewAccessibility::Override* functions and the custom_data_ since any View will be able to set the properties on views it owns.

5. The View's AXNodeData would be kept up-to-date, and we would not need to calculate the values on the fly.

6. We can eliminate bugs caused by GetAccessibleNodeData returning early (and as such not providing values for properties we might need).

    a) 1363612 - There is no way to remove an overridden value from ViewAccessibility's custom_data_ - chromium

    b) 1381484 - Some context menu items lack PosInSet and SetSize information - chromium

How we're going to achieve this

To complete the work proposed here, a lot of code changes will be needed. In early 2023, Joanmarie (jdiggs@igalia.com) began the process of implementing the accessible properties using in part an AXNodeData cached in the base View. All of our ideas revolve around this "cache" for the accessibility content. Here are some merged CLs from the work that Joanmarie started:

- Add the most-commonly-used accessible attributes as View properties which added description and role as accessible properties on View.
    - Ever since this landed, Views have slowly been updated to use SetName and SetDescription rather than their respective Override* functions.
- Ensure IconView views set their accessible name
- Begin using SetAccessibilityProperties in views/controls
- Use AdjustAccessibleName in ImageViewBase; remove GetAccessibleName override
- Remove ViewAccessibility::OverrideDescribedBy
- Begin migration of Ash chrome/browser views to accessibility-property setters
- Begin migration of Ash shortcut viewer views to new accessibility-property setters

And she also had an in-progress CL which serves a sort of big picture for the proposed change:

- [WIP] Replace GetAccessibleNodeData + "Override" functions with properties

A lot of credit is due to Joanmarie for starting all this work and leaving behind great CLs which have served as the basis for this proposal and WIP CLs.

Our end goals are to get rid of the Override* functions, getting rid of the ViewAccessibility :: GetAccessibleNodeData function, and to update the accessible data in the cache as soon as the content update happens.

Challenge to address

This project poses a challenge since we want to prevent Views authors from using the old system (using the Override* methods + GetAccessibleNodeData override approach) on Views, and instead use the new system (Using the Set* setters when appropriate for the View in question). Moreover, we also want to land scoped CLs so it's easy enough to review, revert if necessary, and parallelize among multiple contributors.

In short, we want to parallelize this project as much as possible while preventing two main things:

- Regressions.
- Ensuring that new changes made by Views authors don't move us back.

Proposed Approach

The end state is to update all views to start using these accessible properties, shifting from a "pull" to a "push" model.

The approach we propose is to initially "upgrade" per property and remove the ViewAccessibility::Override* functions as we do so. This makes it so the CLs are not too long and thus easy to review. Moreover, by removing the relevant Override* method we prevent authors from using those functions and forcing them to at the very least use the introduced Set*.

I've started some exploratory CLs that accomplish this as preliminary steps:

- [ViewsAX] Phase out OverrideIsIgnored and OverrideIsLeaf
- [ViewsAX] Phase out OverrideIsEnabled for View::SetIsEnabled
- [ViewsAX] Phase out OverrideName and OverrideRole

These are just exploratory and not final, since in reality we want the CLs to be as small as possible.

So for each property, we'd do on the order of *n* CLs where *n* is the number of Views that use said property.

1. 1st CL would introduce the Setter/Getter for the appropriate property on the ViewAccessibility class.

2. Subsequent CLs would modify Views that use said property to use the new Setters/Getters.
    a. This includes any that are using View::SetAccessible*
3. The *n*th CL would remove the Override* function (if there is one).

Once a given GetAccessibleNodeData override is using only Set* methods, the view will be modified from “pull” to “push”, meaning GetAccessibleNodeData override will be removed altogether and the Set* methods it was calling should be moved to wherever it is appropriate for that view and that property.

It must be noted that not all properties have a respective Override* function in Views, only some do. For example, ax::mojom::StringAttribute::kName has a respective OverrideName function, but ax::mojom::CheckedState does not. Please refer to the “Resources” section for a (non-exhaustive) list of such properties and methods.

Implementation Details

In order to combat the potential issue of authors using an Override* method or a GetAccessibleNodeData override for a property that we have already migrated to the Set* system, as well as to maintain both “push/pull” methods active as we migrate to the new system, this is our proposed structure which modifies the already existing ViewAccessibility::GetAccessibleNodeData:

$let\ customAXND\ =\ existing\ AXNodeData\ cache\ in\ ViewAccessibility\ used\ by\ the\ Override\ *\ methods.$
$let\ axND\ =\ AXNodeData\ cache\ that\ will\ be\ moved\ from\ View\ to\ ViewAccessibility\ (per\ recommendation).$
$let\ nodeData\ =\ the\ out\ parameter.$
$let\ [overrideSteps]\ =\ the\ current\ code\ that\ applies\ the\ Override\ *\ method\ system\ and\ logic\ to\ nodeData.$
$let\ migratedProperties\ =\ a\ set\ containing\ the\ properties\ that\ have\ already\ been\ migrated\ to\ the\ Setters.$
$let\ unionNodeData\ =\ be\ a\ function\ we\ introduce\ which\ "unions"\ two\ given\ AXNodeData\ objects.$

$ViewAccessibility::GetAccessibleNodeData(nodeData)\ \{$
    $view\ ->\ GetAccessibleNodeData(nodeData);$
    .....
    .....
    $[overrideSteps];$
    .....
    .....
    $if\ DCHECK\ is\ On:$
        $for\ prop\ in\ migratedProperties:$
            $DCHECK(prop\ is\ not\ in\ nodeData);$
    $unionNodeData(axND,\ nodeData);$

}

This way, an author that is trying to use either of the old systems to expose an already migrated accessible property will be able to locally realize that it is deprecated and should use the new system. This approach will also ensure that as we migrate properties, the properties that are not yet migrated can still be exposed using the old system and thus both systems can coexist while the migration is in progress.

Resources

Here is a list of the Override* functions that currently exist. We could already start removing them and changing to use Set* counterparts:

- [ ] OverrideIsIgnored
- [ ] OverrideIsEnabled
- [ ] OverrideRole
- [ ] OverrideName
- [ ] OverrideDescription
- [ ] OverrideIsLeaf
- [ ] OverridePosInSet
    - [ ] ClearPosInSetOverride
- [ ] OverrideBounds
- [ ] OverrideHasPopup
- [ ] OverrideLabelledBy
- [ ] OverrideIsSelected
- [ ] OverrideNextFocus

There are other properties that are commonly set by Views' GetAccessibleNodeData method but are not as widespread as the Override* properties, which we could make setters for, like we would for the Override* methods above. Among these:

- [ ] SetClipsChildren
- [ ] SetActiveDescendant
- [ ] SetAutocompleteType
- [ ] SetHasPopup
- [ ] SetIsChecked
- [ ] SetIsDefault
- [ ] SetIsEditable
- [ ] SetIsExpanded
- [ ] SetIsHovered

- [ ] SetIsModal
- [ ] SetIsMultiselectable
- [ ] SetIsProtected
- [ ] SetIsReadOnly
- [ ] SetIsSelected
- [ ] SetIsVertical
- [ ] SetLiveRegionStatus
- [ ] SetPlaceholderText
- [ ] SetPopupView
- [ ] SetPopupFor
- [ ] SetSelectedRange
- [ ] SetTextInputType
- [ ] SetRowCount
- [ ] SetColumnCount
- [ ] SetAriaRowCount
- [ ] SetAriaColumnCount
- [ ] SetMaxValue
- [ ] SetMinValue
- [ ] SetStepValue
- [ ] SetNumericValue
- [ ] SetIsScrollable
- [ ] SetScrollX
- [ ] SetScrollXMin
- [ ] SetScrollXMax
- [ ] SetScrollY
- [ ] SetScrollYMin
- [ ] SetScrollYMax
- [ ] SetSelectionContainer

*This list is not exhaustive, since someone can introduce a new attribute in AXNodeData and this list will not be updated.