SNAPCM

By John D. Heintz, Consultant, Innodata Isogen

By Joshua Reynolds, Consultant, Innodata Isogen

SnapCM is an abstract model for a Configuration Management and Version Control system which supports both time-based management of versioned resources and version--aware references. The management of versioned resources and versioned references can be very simple, or scale to arbitrary complexity by predefined extension points.

SnapCM provides the necessary support for systems that manage the lifecycle of linked resources. Examples include hyper-document lifecycle management, parts database, workflow and literate programming systems.

SnapCM describes the individual Versions of a set of Resources over time. Version-to-Version associations must be handled by versioning-aware References objects that can resolve (via extensible policy) the correct target Version. Versions that exist at the same time and are used in the same context are effective together this is referred to as a Snapshot of those Versions. A consecutive sequence of Snapshots forms a Branch. A Branch also represents the context for effective Versions, allowing multiple Versions (of the same Resource) to be effective at the same time in parallel Branches. Finally, a Sync exposes Versions (effective and historical) from one Snapshot (and contextual Branch) to another.

1.1 Introduction and Goals

This paper defines a number of abstract data types that together form a foundation for describing versioning systems. As an abstract model only those characteristics that are essential are described (specified) and everything else is left unsaid (unspecified). This guideline means that the resulting model is small, precise, and uncluttered with implementation details.

A useful versioning system needs to:

- Enable capturing as much information about versioning actions as possible. That is, one purpose of a configuration management system is to remember what was done and by whom.
- Enable sets of versions to be manipulated within atomic transactions.
- Enable management of local working spaces ("sandboxes").
- Enable both pessimistic and optimistic locking.
- Enable hyperdocuments*[Literary Machines]* and Referent Tracking Documents*[RTD]* to be implemented with the *SnapCM* model with minimum overhead.

Other versioning systems like *[CVS]* and *[WebDAV]* exists but don't meet some or all of the outlined goals.

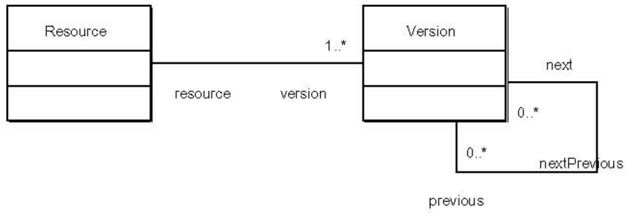
Note: The designs specified in this document are defined using the *[UML]*, and particularly using OCL[1]. The OCL specifications, while provided inline, are also informally described in the text so readers can safely skip over the OCL invariants without losing information.

```
Uppercase names, like "Version", refer to types defined in this paper.
       Lowercase names, like "version", refer to common usage of the name
       (based on context).
```

1.2 SnapCM Model

1.2.1. Resources and Versions

Figure 1. Resource-Version relationships



The relationships between Resources and Versions is one to many.[2] Versions are related to other Versions through many next and many previous properties.

All Versions that share next/previous relations must share the same Resource object. In OCL:

```
context Version
inv same_resource :
       self.previous>notEmpty implies
self.previous>forAll(resource = self.resource) and
self.next>notEmpty implies
self.next>forAll(resource = self.resource)
```

Figure 2. Versions Object Model.

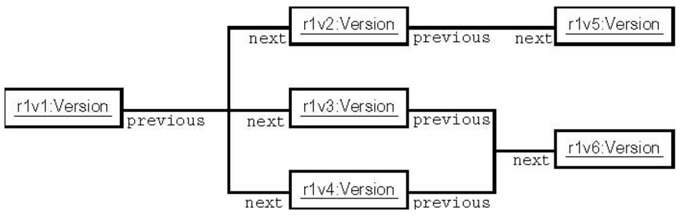
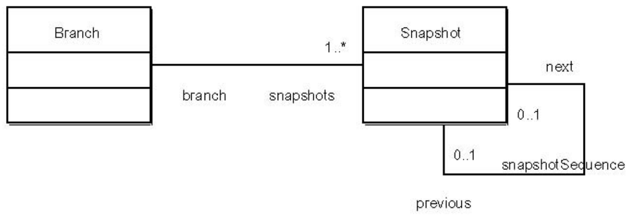


Figure 2 shows an example object model for Versions. We see that individual Versions can have 0..* next and previous associations.

Note: As specified in the Versions same_resource invariant, these Versions must all share the same Resource instance.

1.2.2. Branches and Snapshots

Figure 3. BranchSnapshot relationships.



One Branch is related to an ordered list of Snapshots.[3] Snapshots are related to each other through optional single next and previous properties.

All Snapshots that share next/previous relations must share the same Branch object. In OCL:

```
context Snapshot
inv same_branch :
self.previous>notEmpty implies
       self.previous.branch = self.branch and
self.next>notEmpty implies
       self.next.branch = self.branch
```

The snapshotSequence association does not exhibit the multiplicity that the nextPrevious (see Figure 2) association does. This means that Snapshots are only directly related to each other as a sequence and doesn't explain how merging or branching behavior is accomplished. This is explained later in the description of the Sync structure.

Figure 4. Snapshots Object Model.

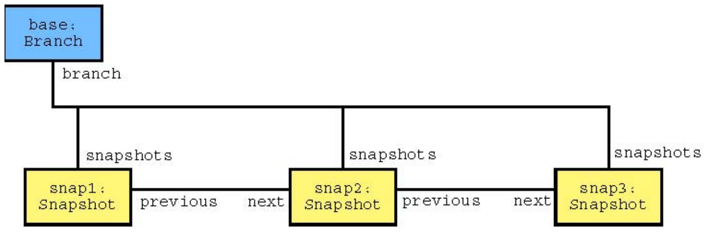
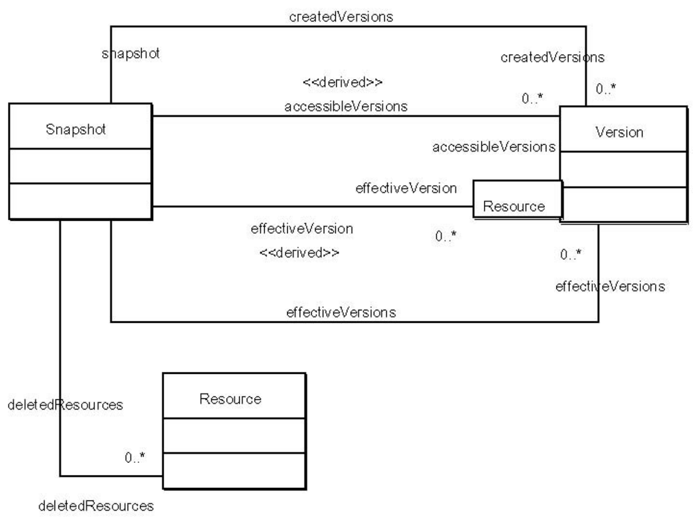


Figure 4 shows a single Branch with three Snapshots.

1.2.3. Snapshots, Versions, and Resources

Figure 5. Snapshot-Version-Resource relationships.



Snapshots have several important relationships with Versions and Resources. These relationships are used to expose both which Versions (and therefore Resources) are present on a given Snapshot as well as which Versions could be present on the "next" Snapshot.

createdVersions

```
The "createdVersions" association identifies all Versions that were
       created on this Snapshot. This is the first place in the model that
       a new Version instance becomes available for use only Versions
       with a "snapshot" property may have associations between Snapshots
       and Versions.
```

effectiveVersions

The "effectiveVersions" association identifies all Versions that are visible on this Snapshot.[4] This association is constrained to only include Versions that are also accessible on this Snapshot and not otherwise hidden. In OCL:

context Snapshot

inv limit_effectiveVersions :

self.accessibleVersions>includesAll(self.effectiveVersions) and

self.effectiveVersions.resource>excludesAll(self.hiddenResources)

Additionally, effectiveVersions may only contain one Version for a given Resource. In OCL:

```
context Snapshot

inv one_version_effective :

--for each effectiveVersion.resource

self.effectiveVersions.resource
```

```
    ->forAll(r |
            self.effectiveVersions->select(v |
                    v.resource = r)->size < 2)
```

effectiveVersion

This qualified association provides a simple means of navigating from a Resource to a Version on a particular Snapshot. The navigation is defined in terms of the effectiveVersions property of a Snapshot. In OCL:

```
context Snapshot

inv define_effectiveVersion :

self.effectiveVersion = self.effectiveVersions>select(v |

    v.resource = resource)
```

accessibleVersions

The "accessibleVersions" association defines accessiblity in terms of history (self.previous.accessibleVersions), creation (self.createdVersions), and syncing (self.syncSources.accessibleVersions). In OCL:

```
context Snapshot

inv define_accessibleVersions :

self.accessibleVersions =

    self.previous.accessibleVersions

            ->union(self.createdVersions)

            ->union(self.syncSources.accessibleVersions)
```

There is no restriction on accessibleVersions like the one_version_effective invariant for effectiveVersions; accessibleVersions can contain many Versions from a single Resource.

hiddenResource

the "hiddenResource" association exists to identify those Versions by Resource that are not allowed to be present in the effectiveVersions property of Snapshots. This property has no effect on which Versions are accessible, and there is no effect on previous Snapshots effectiveVersions.

A Resource may be present on the hiddenResources property only if one or more of its Versions are accessible on the Snapshot. In OCL:

```
context Snapshot

inv limit_hiddenResources :

self.accessibleVersions.resource>includesAll(self.hiddenResources)
```

Figure 6. Snapshot-Resource-Version Grid.

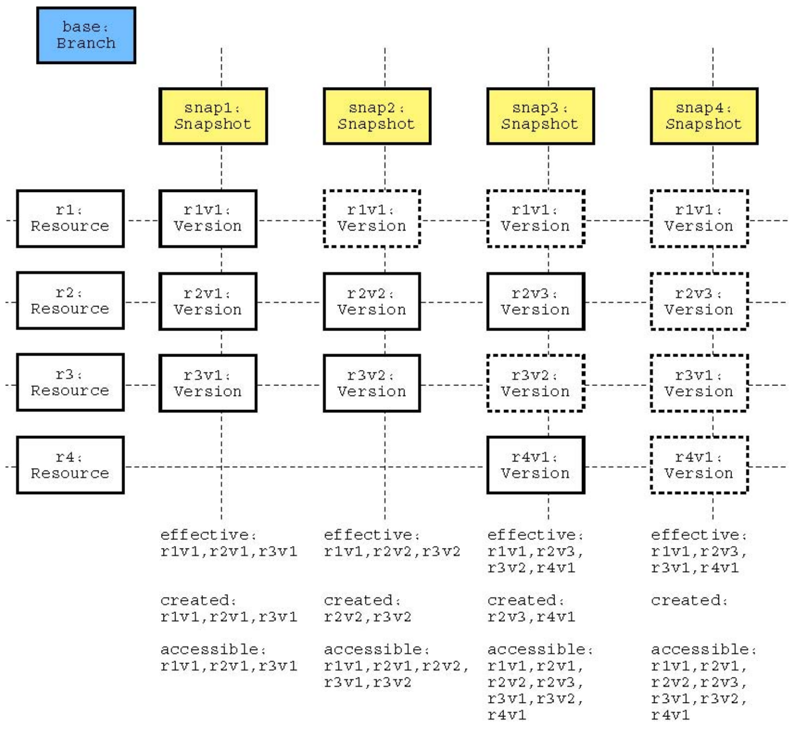


Figure 6 shows an instance-like diagram of Snapshots, Resources, and Versions. Horizontal lines represent Versions of a single Resource. Vertical lines represent effective Versions on a Snapshot. Note that the sets of effectiveVersions, accessibleVersions, and createdVersions are listed under each Snapshot.

Versions instances are shown under Snapshots with two different graphic border styles to show how they are related to the Snapshot.

Solid

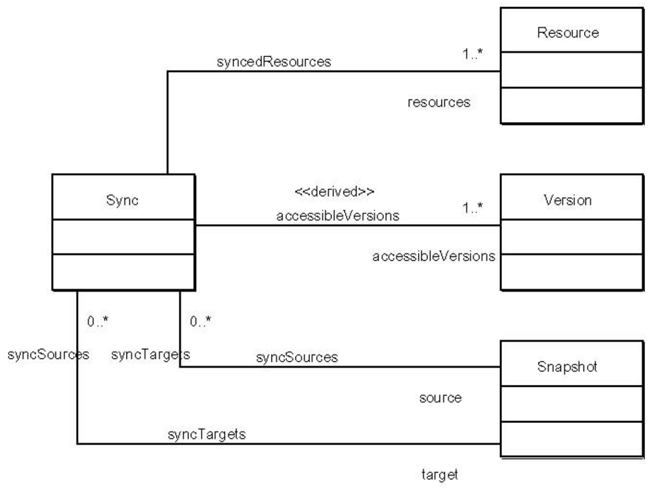
```
Versions with solid line borders are created on the Snapshot vertically
       above them.
```

Dotted

```
Versions with dotted line borders represent Versions that are effective
       on the Snapshot vertically above them, but not created on that
       Snapshot.
```

1.2.4. Syncing Versions between Snapshots

Figure 7. Sync-Snapshot-Resource relationships.



The concepts of branching and merging are reified in the Sync type. Branching and merging are accomplished by expanding the accessible Versions available on some target Snapshot (and therefore the owning Branch). Examples of how to accomplish branching, merging and publishing with Syncs are given after descriptions of the associations between Syncs and Snapshots, Resources, and Versions.

syncSource

```
The "syncSource" association identifies by Snapshot the accessible
       Versions (and therefore Resources) that can participate in the
       Syncing.
```

syncTarget

```
The "syncTarget" association identifies the Snapshot whose accessible
       Versions set is expanded as a result of Syncing.
```

syncedResources

```
The "syncedResources" associations identifies for the Sync the total set
       of Resources that contribute accessible Versions from the source
       Snapshot to the target Snapshot. A Resource may be a member of the
       Sync resources property only if a Version for that Resource is
       accessible on the source Snapshot. In OCL:

       context Sync

       inv limit_resources : self.source.accessibleVersions.resource
       ->includesAll(self.resources)
```

accessibleVersions

```
The "accessibleVersions" association is derived from the source and
       resources properties as all Versions accessible from the source
       Snapshots whose Resource is a member of the Sync's resources
       property. In OCL:

       context Sync

       inv define_accessibleVersions :

       self.accessibleVersions =self.source.accessibleVersions

              ->select(v:Version | self.resources>includes(v.resource))
```

Common Syncing Forms

Branching

Branching is a forking of some set of Versions (accessible from the source Snapshot). This is typified by a Sync from some source Snapshots to a new Branch.

Figure 8. Sync Branching.

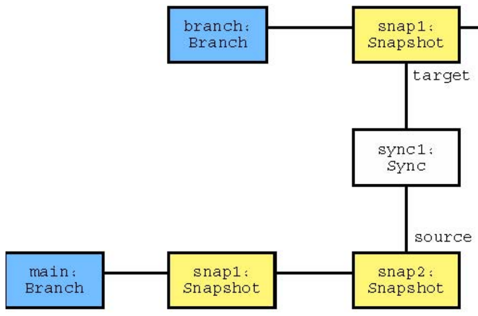


Figure 8 shows an example of Syncing to achieve a branch. Notice that the source Snapshot (main:snap2) exists on a Branch with some history (here only main:snap1) while the target Snapshot (branch:snap1) is the first Snapshot on its Branch.

With this structure further modifications can occur on branch:Branch without disturbing the Versions on main:Branch.

Merging

Merging is reintegrating parallel changes back into some original Branch. This is typefied by a Sync with transposed source.branch and target.branch properties and shared Resources from any other Sync.

Figure 9. Sync Merging.

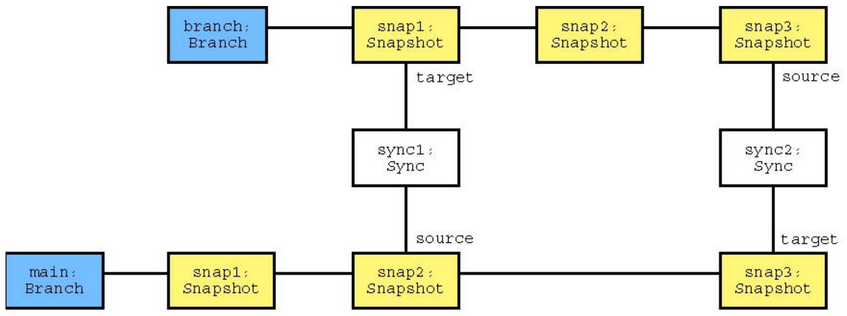


Figure 9 shows an example of Syncing to achieve a merge. Notice that the two Syncs have inverse source.branch and target.branch properties. Assuming that at least some of the Resources from sync1:Sync are also present on sync2:Sync this merges previously branched Resources back into main:Branch.

Sync Masking

Sync Masking provides a controlled exposure of effective Versions from a given Branch. This is typefied by a list of Syncs with the same source.branch and target.branch properties, with the further condition that no Snapshots are created on the target Branch except by Syncs from the list.

Figure 9. Sync Merging.

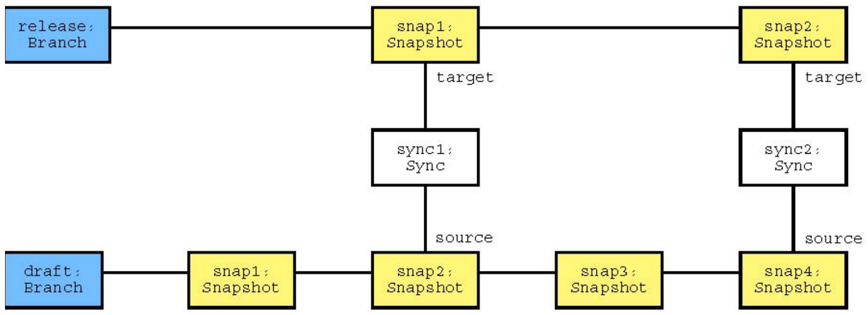


Figure 10 shows a stereotypical Sync Masking example. Note that the target Snapshot (release:snap1) effectiveVersions is a subset of the source Snapshot (draft:snap2) effectiveVersions, and that the target Snapshot (release:snap2) effectiveVersions is a subset of source Snapshot (draft:snap4) effectiveVersions. Also, notice that no Snapshots exist on release:Branch except for those Synced from draft:Branch.

Sync Example

Figure 11. Syncing Example.

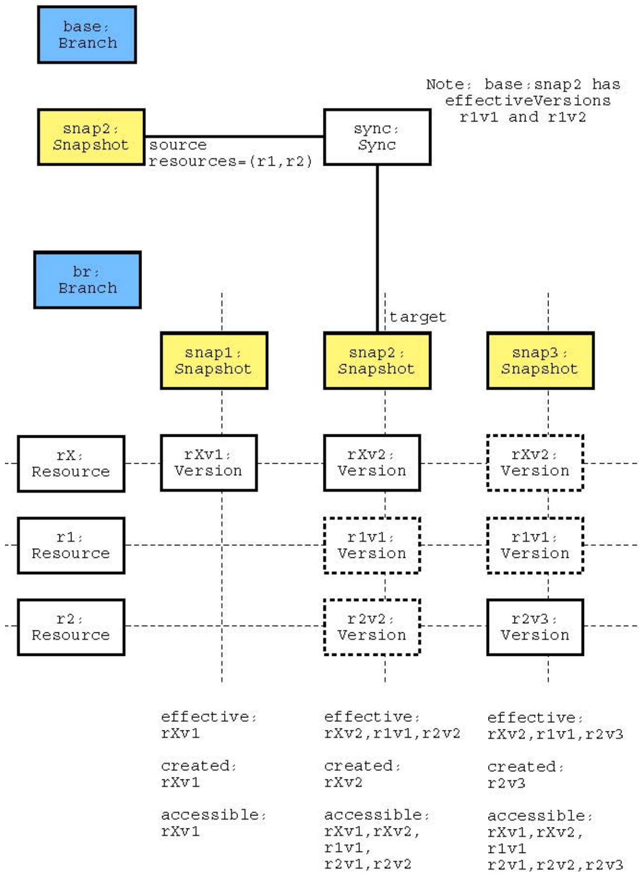


Figure 11 shows a detailed example of Syncing which builds on the diagram from Figure 7. Note that:

- Resources r1 and r2 were not accessible on br:Branch before the Sync occurred at br:snap2.
- r2v3 is not accessible on base:Branch, only on br:Branch.
- r1v1 and r2v2 are not copied into br:Branch, but rather are shared references by both Branches.

References from Versions to Resources

Figure 12. References

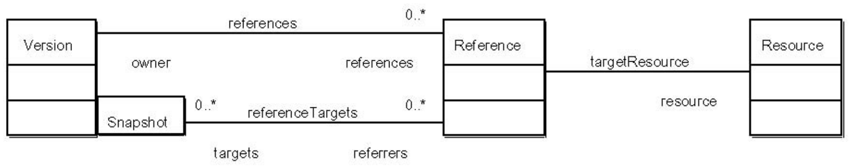


Figure 12 shows how Versions use References to navigate from Resource to Versions given a resolution Snapshot. References are essentially version-aware pointers.

references

```
The "references" association identifies the set of References owned by a
       given Version.
```

targetResource

```
The "targetResource" association identifies the target of the Reference.
       The Resource identified may not be the owner Version's Resource. In
       OCL:

       context Reference

       inv no_self_references :

       elf.owner.resource <> self.resource
```

referenceTargets

```
The "referenceTargets" association identifies the resolved Versions of the
       targetResource. Resolution is always qualified by a Snapshot.
```

Figure 13. Reference Resolution Policy.

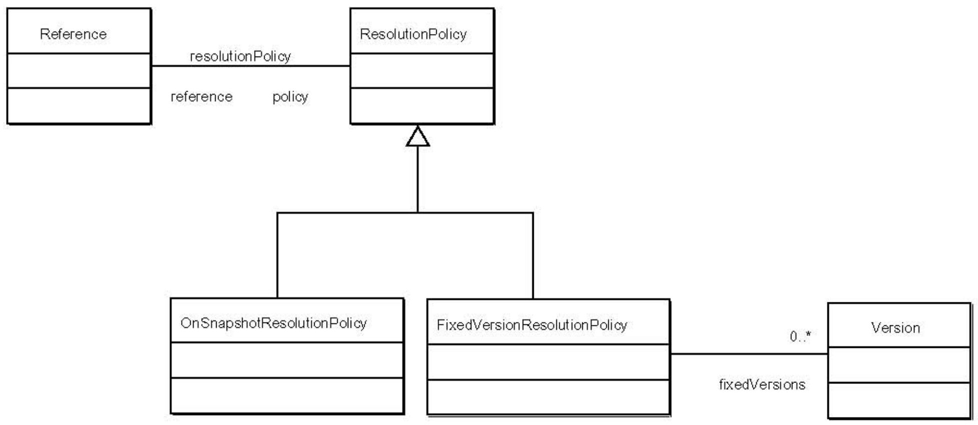


Figure 13 shows that a Reference has a ResolutionPolicy. The ResolutionPolicy specifies the logic for resolving the Reference.targets property using the Reference.targetResource property and the snapshot:Snapshot qualifier.

resolutionPolicy

> The "resolutionPolicy" assiociation identifies the ResolutionPolicy for a Reference which is used to calculate the Reference.targets property.

There are two predefined types of ResolutionPolicies that specify very common resolution behavior.

OnSnapshotResolutionPolicy

```
This policy defines the Reference.targets property to be equal to the
       Version that is effective on the qualifying Snapshot. In OCL:
       context OnSnapshotResolutionPolicy
       inv targets_inv :
       let snapshot = self.reference.referenceTargets.snapshot in
          self.reference.targets =
               snapshot.effectiveVersion(self.reference.resource)
```

FixedVersionsResolutionPolicy

> This policy defines the Reference.targets property to be equal to some arbitrary set of Versions. In OCL:

```
context FixedVersionsResolutionPolicy
inv targets_inv :
self.reference.targets = self.versions
```

Figure 14. Reference Example.

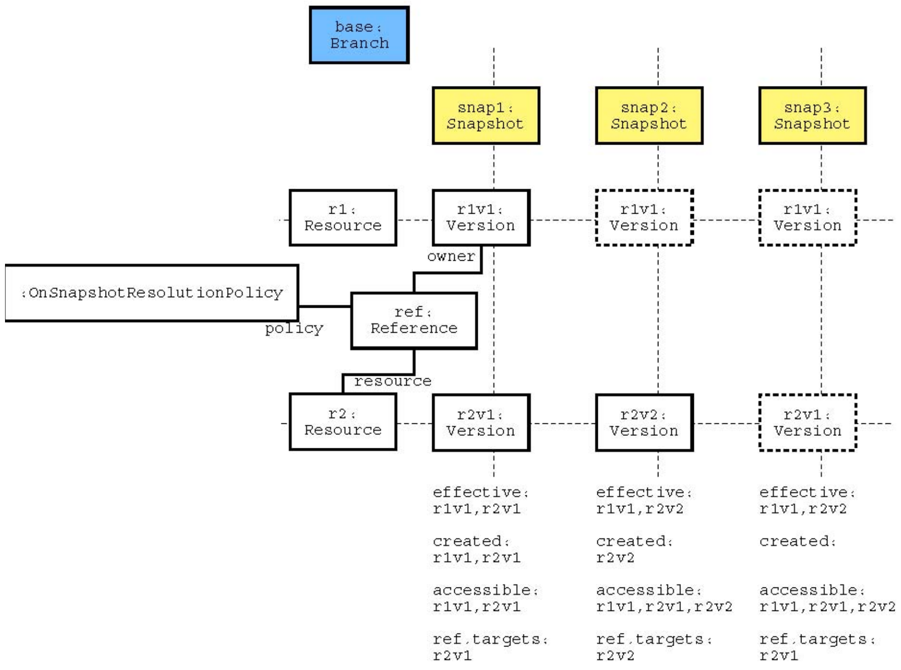


Figure 14 shows an example use of a Reference with an OnSnapshotResolutionPolicy. Note that the resolved d1.targets property is listed under each Snapshot.

Organizers for Resources

Figure 15. Organizer Model.

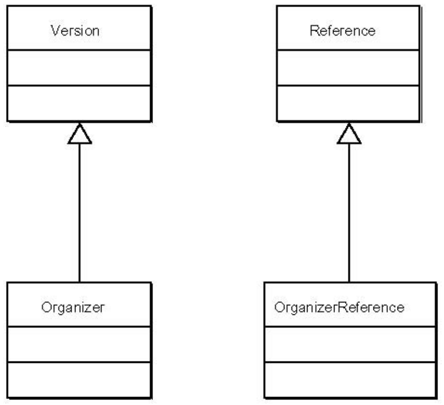


Figure 15 shows built-in subtypes of Version and Reference useful for logical organization of Versions. By using OrganizerReferences, the resultant structure is inherently version-aware. Using a Reference subtype enables easier Version.referrers management.

Figure 16. Organizer Example.

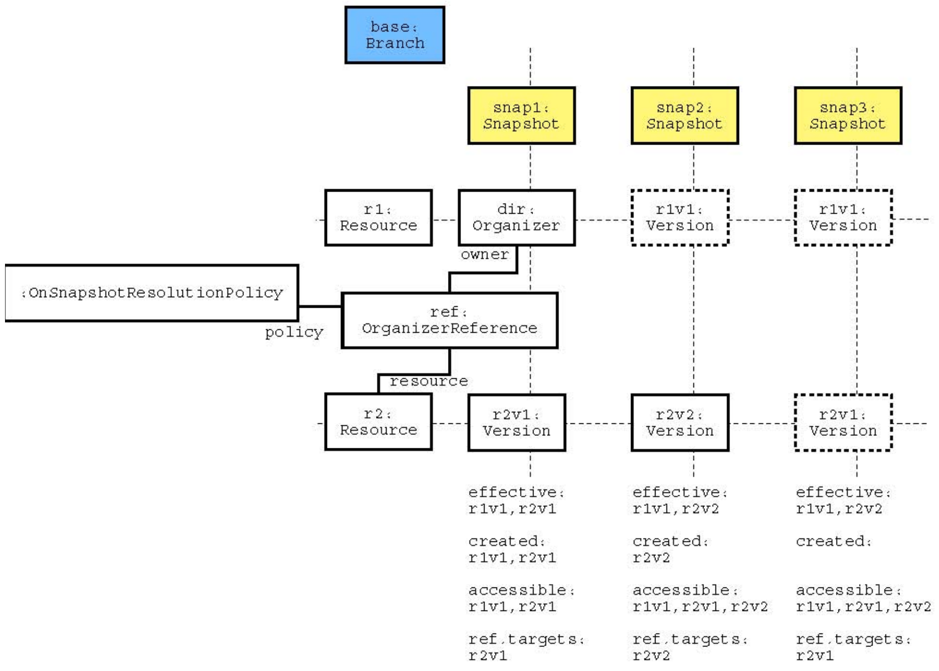


Figure 16 shows an example of using Organizer to model a directory structure. Note that the diagram looks exactly like Figure 14 except for the subtypes Organizer and Organizer Reference are used.

1.3 Conclusion

SnapCM defines a coherent model for describing version control and configuration management problems and solutions. Sequences of Snapshots stases version history needs, Synchronized Branches satisfy parallel evolution needs, and References with ResolutionPolicies satisfy version-aware linking needs. Further work includes refining for specific problem domains.

1.4 Bibliography

[UML], Online from **http://www.uml.org**

[Literary Machines] Nelson, Theodor H. 1987. *Literary Machines*. South Bend, Indiana: The Distributors.

[RTD] W. Eliot Kimber, Peter Newcomb, Steve Newcomb. "Version Management as Hypertext Application: Referent Tracking Documents." Online from **http://www.isogen.com/papers/ref-track-docs-paper.pdf**

[CVS], Online from **http://www.cvshome.org**

[WebDAV], Online from **http://www.webdav.org**

1.5 Notes

The models in this paper use several UML diagrams and figure types. The types of diagrams found in this paper are the Class diagram, the Instance diagram, and a modified (grid shaped) Instance diagram optimized for presentation of Versions on Snapshots.

The Class diagram is used to describe types (or classifications) of objects. The Generalization relationship (with an open triangle on top) is used to specify subtypes. The Association relationship is used to specify properties and navigation. Sometimes an Association has a black diamond to represent a composite relationship to signify ownership/containment. Additionally, Associations can have a qualifier (type) at one end to specify qualified properties. An example of qualification is User associating to Account qualified by AccountID.

The Instance diagram is used to show example objects from the Class diagrams. Instances are named and typed.

Finally, the Grid Instance diagrams are like Instance diagrams except they imply addition instance relationships based on vertical alignment with a Snapshot instance and box type. Version instances lined up vertically on a Snapshot instance are all in the Snapshot's effectiveVersions property set. Additionally, Version instances that have solid box lines (instead of dotted) are in the Snapshot's createdVersions property.

2. A Resource could be described as a set of Versions (those related by next/previous properties) that is set exclusive with all other Resource sets.

3. A Branch could be described as a sliding temporal view over the intersections of Resources and Snapshots.

4. A Snapshot could be describes as a set of effective Versions.

About Innodata Isogen

Innodata Isogen (**www.innodata-isogen.com**) optimizes content supply chains, helping clients realize significant cost savings and productivity gains from operations, achieve better outcomes and compete more effectively in demanding global markets.

Our solutions encompass virtually every activity necessary to create, use and distribute information products. Clients can choose from an array of integrated point solutions or simply outsource their entire content supply chain to us.

The company, based in metro New York, has seven content solution centers and satellites in North America and Europe, six certified content production facilities and a dedicated tools and technology center in Asia, together employing more than 7,000 content specialists worldwide.

Among our extensive roster of blue-chip clients are the world's leading commercial publishers, Global 2000 enterprises, government agencies, and major archives, libraries and museums.

In 2001, *Fortune* and *Business Week* each rated the company as one of the 100 top-performing public companies in the U.S.

Contact Innodata Isogen today to find out how we can help you more effectively manage your organization's information assets:

Corporate Headquarters
Innodata Isogen
Three University Plaza
Hackensack, NJ 07601
U.S.A.
T (201) 488-1200
T (800) 567 4784 – (toll free)
F (201) 488-9099

solutions@innodata-isogen.com
http://www.innodata-isogen.com