Maintaining Consistency Among Observers: An Alloy Model of the Observer Design Pattern

Roshan Gupta

Abstract

The goal of this project is to explore how to ensure that two or more observers viewing the same object always reflect a synchronized view of that object's state. Three variants of the Observer design pattern were modeled using Alloy, each placing different degrees of restriction on the mapping between subjects and observers. In addition, two notions of consistency were also modeled. *Partial Consistency* implies that two observers are updated in the same order, and *Complete Consistency* adds to this the fact that two observers are updated the same number of times. The results indicate that the most restrictive variant is always partially consistent by definition, and that all three variants can be made both partially and completely consistent with the introduction of either local or global lock tables.

1. Introduction

The goal of this research project is to explore how to maintain data consistency between multiple observers in a system based on the Observer design pattern. I seek to enumerate the minimum constraints that must be imposed on such a system to ensure that two or more observers viewing the same subject always reflect a synchronized view of that subject's state.

1.1 Motivation

My Master's thesis is to design a tool that records, manipulates, and plays back ETMS data feeds to be used as test feeds to air traffic control applications. It makes sense that users of the tool should be able to view the information in a feed in multiple ways. For example, a user might wish to simultaneously view the current locations of all planes both visually (on a two-dimensional map) and in table format. In addition, the user might make a modification in one of the views—say the table format—and expect the change be automatically reflected in the other view. Hence, it is important that both views always display a consistent and synchronized view of the data.

The Observer design pattern is a good model for this program. It allows you to define different display modules that are decoupled from each other yet act as if they are synchronized when they view the same object. It also gives you the flexibility of dynamically adding and removing views of the data without the need to modify and recompile your code. The challenge however is to maintain synchronization between observers as the mapping between subjects and observers becomes more complex. There may be instances where it is desirable to have observers view multiple subjects—but not necessarily the same set of subjects between observers. For example, the table view in the ETMS Feed Parser (EFP) might need to observe and integrate multiple feeds of data—say, input from two different radar sources. The visual view might also observe these two input streams, but in addition observe a weather feed to allow it to predict the trajectories of each plane and display both the actual and predicted trajectories on a map. If both these views happen to be active, it is important that changes in the two radar feeds get immediately propagated to both views to prevent the visual view from predicting trajectories based on stale data.

2. Definitions

2.1 Observer Pattern Topologies

There are three different variants of the Observer pattern that are of interest; they differ with respect to the allowable subject-observer mappings (topologies).

Most-Restrictive Topology

In this variant of the Observer design pattern, observers are constrained to view at most one subject. Multiple observers are allowed to view the same subject. Figure 1 is an example of an acceptable topology.

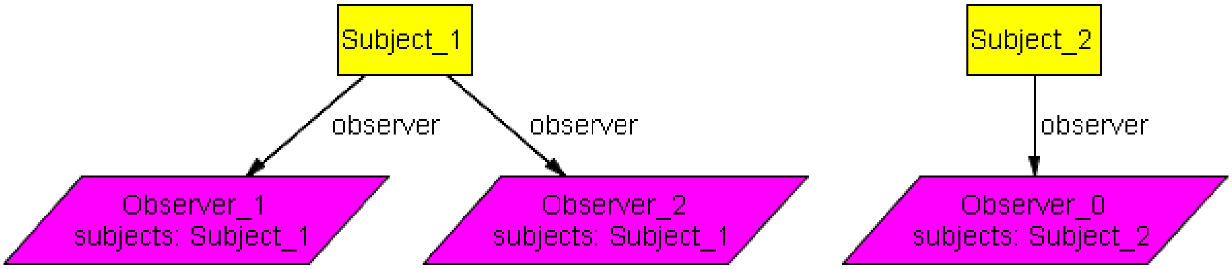

Figure 1: An example of the Most-Restrictive topology.

Semi-Restrictive Topology

In this variant of the Observer design pattern, observers are allowed to view a set of subjects. That is, the subjects in the system are partitioned into one or more distinct sets, and each observer can view the subjects in at most one of these sets. Multiple observers are allowed to view the same set of subjects. Figure 2 is an example of an acceptable topology.

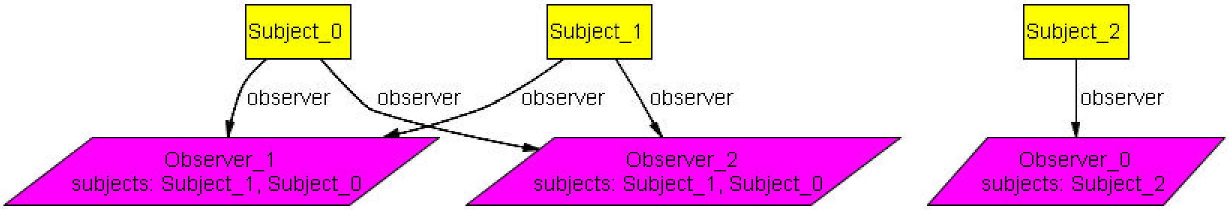

Figure 2: An example of the Semi-Restrictive topology.

Least-Restrictive Topology

In this variant of the Observer design pattern, observers are allowed to view an arbitrary number of subjects, independent of other observers. There are essentially no restrictions on the subject-observer mapping. Figure 3 is an example of an acceptable topology.

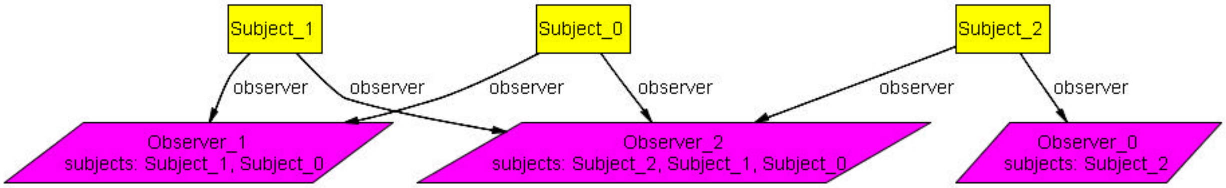

Figure 3: An example of the Least-Restrictive topology.

2.2 Consistency

I use two different notions of consistency in this experiment—*complete consistency* and *partial consistency*. For the following two subsections, assume that there exist two observers, `O` and `P`. Each observer has a set of subjects it is monitoring, and each observer maintains a history of the notifications it has received. For example, observer `O` might look like this:

```
O’s subjects = {S1, S2, S3}
O’s notification history = {S1, S3, S2, S1}
```

In this example, `S1`, `S2`, and `S3` represent three distinct subjects. Observer O’s notification history indicates that O was first notified of a change in subject `S1`, then `S3`, then `S2`, and most recently `S1` again.

Complete Consistency

Complete consistency implies that two observers are perfectly synchronized with respect to the state information they maintain about the subjects they are jointly viewing. Two observers are completely consistent if they have both been notified of changes the same number of times and in the same order by their common subjects. See the text box below for an example. Ideally, we would like to achieve a system in which every pair of observers is guaranteed to be completely consistent at any point in time.

Examples Of Complete Consistency

Here is an example where observers `O` and `P` are completely consistent (note that both `O` and `P` have been notified by `S1` and `S2` in the same order `{S1,S2,S1}`):

```
O’s subjects =
```

Partial Consistency

Partial consistency makes a weaker statement compared to complete consistency—it only implies that two observers have received change notifications from their common subjects in the same order. This guarantees that two observers will always show a mutually consistent view of the data, though one observer's data might reflect an earlier state of the system. This notion of consistency might be acceptable to an application that emphasizes state coherence over the timeliness of data (for instance, an air traffic control application is often more interested in providing a valid snapshot in time of an airspace region than in providing an inconsistent view of the current air traffic).

Two observers are partially consistent if they have both been notified of changes in the same order (but not necessarily the same number of times) by the set of subjects being monitored by both. See the text box below for an example.

3. Methods

I modeled a system based on the observer design pattern using the Alloy modeling language. I was then able to use Alloy's automatic analyzer to explore what kinds of constraints are necessary to achieve various levels of consistency among the observers in the system.

It is not straightforward that Alloy is the best choice to model this problem. The idea of maintaining consistency amidst change requires one to take time into account, and Alloy is ideal for modeling structural complexity rather than temporal complexity. However, Alloy has the advantage in that I can say as little as I want to about the system and incrementally add constraints, whereas in a prototype based on a programming language I would have to build a larger infrastructure to allow for more possibilities. In the end, I decided it was worth the extra effort to model the temporal aspects in Alloy in order to gain the benefits of Alloy's compact representation.

3.1 Alloy Model Summary

The actual Alloy model can be found in the Appendix. The model is well-organized and thoroughly documented.

The approach I took was to initially model the Least-Restrictive variant of the Observer

Examples Of Partial Consistency

Here is an example where observers `O` and `P` are partially consistent (note that this is the same example as the example above where `O` and `P` were not completely consistent):

```
O's subjects = {S1, S2, S3}
P's subjects = {S1, S2, S4, S5}
O's notification history = {S1, S3, S2, S1, S3}
P's notification history = {S4, S1, S2, S5}
```

Here is an example where observers `O` and `P` are *not* partially consistent (note that the only difference between this example and the previous one is that `S2` notified `P` before `S1` was able to):

```
O's subjects = {S1, S2, S3}
P's subjects = {S1, S2, S4, S5}
O's notification history = {S1, S3, S2, S1, S3}
P's notification history = {S4, S2, S1, S5}
```

design pattern. The remaining two variants could then be easily modeled by adding a few constraints on the system topology (refer to the section **TOPOLOGY FUNCTIONS** in the Alloy model). I focused my efforts on modeling the transmission of notification messages from subjects to observers. I defined two basic signatures, Subject and Observer. Each Observer maintains a history of the notifications it has received from the subjects it is viewing. This history takes the form of a sequence (I utilized the *seq.als* model provided with Alloy), where notifications by subjects are proxied by the subject atoms themselves. See Figure 4 for an example snapshot of a system.

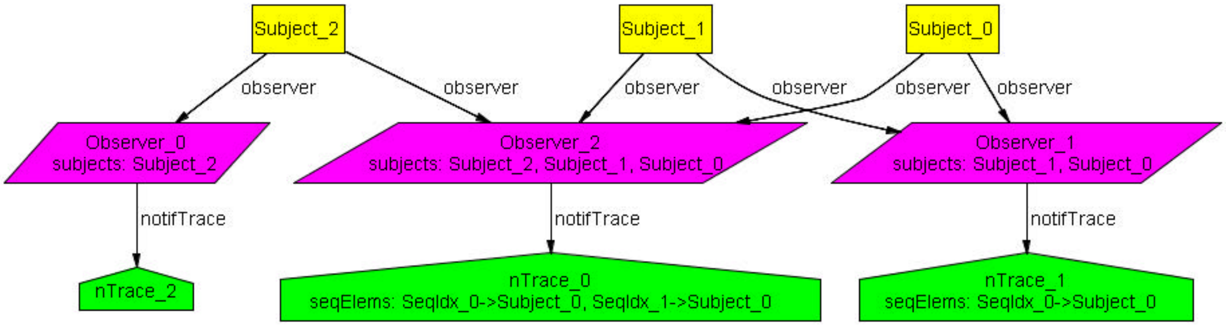

Figure 4: Snapshot of a Least-Restrictive topology system.

In addition to the basic Subject and Observer signatures, the model maintains a mapping between atoms of these types. The subjects, observers, and topology together make up a snapshot of the system's state. I then modeled the basic rules governing valid state transitions (refer to the **STATE TRANSITION RULES** section in the model). The two major rules are `NoTopologyChange`, which prevents the topology from changing, and `AtMostOneNewNotifPerObserver`, which handles the fact that an Observer can handle at most one notification at a time.

Finally, I created two functions `CompletelyConsistent` and `PartiallyConsistent` that determined whether two observers were completely or partially consistent, respectively. These two functions became the basis of the assertions I later made about the system. Refer to the section **CONSISTENCY DEFINITIONS** in the model for more information.

3.2 Assumptions

I simplified the design of the model by making two main assumptions. I first assumed that if an Observer is notified of a change in one of its subjects, it immediately calls *update* on the subject to retrieve its latest state. This allowed me to consolidate the notification and update operations into a single notification operation, and as a result my model only needed to maintain the notification history for a given Observer. This assumption is valid since it is too difficult to maintain consistency among observers if one cannot guarantee that an Observer will react to a change in state.

I secondly assumed that the system topology does not change between state transitions. At first this may seem to rule out the ability of my system to support the dynamic adding and deletion of views. Instead, it makes it simpler to reason about the correctness of the model, and this reasoning can be easily extended to a model that allows for topology changes. In particular, if the static-topology model demonstrates that a system starting in a completely or partially consistent state will always remain consistent, then the dynamic-topology model would simply need to guarantee that a valid topology change maintains this same level of consistency.

3.3 Questions Asked of Model

There were two questions I was interested in answering with my model:

1) What constraints on the system are necessary to guarantee that for all possible sequences of states, every pair of Observers is always partially consistent?

2) What constraints on the system are necessary to guarantee that for all possible sequences of states, every pair of Observers is always completely consistent?

Since I actually modeled three variants of the observer design pattern, I ultimately created a total of six assertions (one per each pair of question and variant). These assertions can be found in section **CONSISTENCY INVARIANTS & ASSERTIONS** of the model.

4. Evaluation

4.1 Results

This project turned into more of a modeling effort of the observer design pattern and notions of consistency rather than an exploration of system constraints. This was because almost all my effort went into creating a viable model of the system. This task was complicated by my lack of experience with Alloy, subtle bugs in my logic, and issues with the analyzer itself. However, I did find some useful information regarding my original questions. In addition, I now have a solid framework from which to launch queries regarding the observer design pattern.

One of the first results of my experiment was that Alloy asserted that the Most-Restrictive topology was by definition always partially consistent. Indeed, since observers are allowed to view at most one subject at a time, two observers viewing the same subject will always be notified in the same order. Note that two observers who are not viewing any common subjects are by definition always partially (and completely) consistent. Of course, the Most-Restrictive topology does not guarantee complete consistency—to achieve this level of consistency for this model, I simply had to add a transition rule that forced a subject who notified one of its observers to notify all of its observers. With this transition rule in place, Alloy asserted that the Most-Restrictive topology would always be completely consistent.

According to the Alloy analyzer, the Semi-Restrictive topology by definition did not guarantee partial consistency. If I have two observers viewing the same two subjects, and both subjects send out a notification, the notifications might reach each observer in different orders. I was able to achieve partial consistency when I again added the rule that a subject must notify either none of its observers or all of its observers. Since the model does not allow Observers to process more than one notification at a time, one subject essentially "muscled out" the others and transmitted its notifications to all Observers (in retrospect, it would have been nice to rework the model to prevent one subject from implicitly dominating the other subjects during a particular state transition). This rule roughly corresponds to the presence of a lock table, in which only one subject wins the right to talk to the observers within a given period of time. With such a lock structure in place, we can now guarantee that all observers viewing the same set of subjects will receive notifications from those subjects in the same order. In fact, we can also extrapolate that this rule would make the Semi-Restrictive topology completely consistent as well. Unfortunately, the analyzer kept crashing before returning with the final answer to this question.

Finally, the Alloy analyzer stated that the same rule used in the Semi-Restrictive system would achieve partial consistency in a Least-Restrictive system. Again, the rule is analogous to a global lock table, where a subject must lock all of its observers before sending out notifications. Even if there are complicated dependency relationships between subjects and observers (as in Figure 3), the locking mechanism would guarantee that notifications arrive in the proper order. This analysis also probably extends to the complete consistency case, but again the analyzer could not finish checking this assertion.

One interesting note is that the Least-Restrictive system seems to require a global lock table, whereas the Semi-Restrictive system would require only local lock tables (because the subjects and observers in the Semi-Restrictive system form partitions). This is an important distinction when performance is of concern, and hence is an example of a useful result that came from directly modeling these various types of systems.

4.2 Retrospection

I had encountered some intractability issues early on in my first version of the model. One reason is that I had originally attempted to model a history of notifications as a List, complete with functions such as list comparisons and filtering operations. However, my functions were defined using recursion, which is not supported by the current version of Alloy. Due to my inexperience with Alloy, I also made the mistake of modeling the topology of the system by storing a subject's observers directly in the Subject signature, and similarly for the Observer signature. Since a given Subject atom points to a fixed Observer atom, and because each Observer atom also points to a fixed sequence of notifications, a simple state transition in which an Observer received a new notification would require a new Subject, Observer, and Sequence atom. In other words, the first version of the model required a large scope for simple transitions. My second model solved this issue by storing the mapping between subjects and observers, and between observers and traces, in a separate signature. Finally, I had problems checking some of my assertions because Alloy would consistently crash before returning an answer.

Overall, I feel that this project gave me invaluable experience in designing large models in Alloy. I felt that it made up for my mistakes on the Elevator problem, where I did not fully understand the power of Alloy and attempted to model my elevator system procedurally rather than declaratively. My impression of Alloy is that it has a place in the design of large software systems. In particular, I felt the ability to incrementally build your model and receive immediate feedback via the analyzer were its key selling points. The version of Alloy I was using was a little unstable, but that is to be expected from a beta release. One feature in particular I would find useful is if the Alloy analyzer provided feedback regarding why a solution cannot be found. The right feedback can differentiate between a simple logic mistake and a generally flawed approach. My hope is that future improvements will allow me to make much more complex models and queries than what is possible right now. In any case, I hope to revisit this problem again and to continue using this tool while conducting research for my Master's thesis.

```
/******************************************************
Roshan Gupta, 5/13/02

This model explores notions of consistency within
the Observer design pattern.


ASSUMPTIONS:
- If an Observer is notified of a change in one of
  its subjects, it immediately calls "Subject.Update"
  to retrieve the latest state.  Hence, we can
  consolidate the notification and update operations
  into a single notification operation.
- The system topology is fixed, i.e the number of and
  mappings between subjects and observers in a given
  system cannot change.  This shouldn't affect the
  correctness of this model with respect to the goals
  we are trying to achieve.

*******************************************************/

module ObserverDesignPattern

open std/seq
open std/ord



//
// SIGNATURES
//


// A subject or publisher object in the design pattern.

sig Subject {}


// An observer or subscriber object in the design pattern.

sig Observer {}


// A snapshot of the state of the observer-pattern system we are modeling.

sig State {
      -- The observers in the system.
      observers: set Observer,

      -- The subjects in the system.
      subjects: set Subject,

      -- List of observers viewing a given Subject.
      subToObsMap: subjects -> observers,

      -- List of subjects being viewed by a given Observer.
      obsToSubMap: observers -> subjects,
      -- The order in which a given Observer has been notified by its
      -- Subjects (a notification is proxied by the actal Subject atom).
      -- Note that the last element in the sequence represents the most
      -- recent notification.
      --
      -- For example, notifTrace = {S1, S3, S2, S1} means that the Observer
      -- first received a notification from subject S1, then S3, then S2, and
      -- most recently S1 again.
      notifTrace: observers ->! Seq[Subject]
}
{
      -- The mapping of observers to subjects is the transpose of the mapping
      -- of subjects to observers.
      obsToSubMap = ~subToObsMap

      -- An Observer can only receive notifications from the subjects it
      -- is monitoring.  This is based on the assumption that the
      -- topology is fixed (i.e. any notification from a particular subject
      -- in its trace implies it has and will always monitor that subject).
      all o: observers | SeqElems(o.notifTrace) in o.obsToSubMap
}




//
// STATE INITIALIZATION FUNCTIONS
//


// Initializes the State's observers' notification traces to empty.

fun InitTracesToEmpty (s: State) {
      all o: s.observers | SeqIsEmpty(s.notifTrace[o])
}




//
// STATE TRANSITION RULES & FUNCTIONS
//


// Ensures Subject-Observer mappings do not change between
// state transitions.
fun Rule_NoTopologyChange (s, s': State) {
      s'.subjects = s.subjects
      s'.observers = s.observers
      s'.subToObsMap = s.subToObsMap
}


// Ensures that each Observer receives at most one new notification
// during a state transition.  Also, ensures that each Observer's
// notification history remains the same.

fun Rule_AtMostOneNewNotifPerObserver (s, s': State) {
      all o: s'.observers | {
            SeqStartsWith(s'.notifTrace[o], s.notifTrace[o])
            #SeqInds(s'.notifTrace[o]) < #SeqInds(s.notifTrace[o]) + 2
      }
}


// Ensures that if a Subject notified one of its observers during a state
// transition, it must have notified all of its observers during that
// same transition.

fun Rule_NotifyOneNotifyAllInSameTransition (s, s': State) {
      all j: s'.subjects | {
            all o: s'.subToObsMap[j] | {
                  receivedNotifFromSubject(s, s', o, j) implies {
                        all p: s'.subToObsMap[j] - o | {
                              receivedNotifFromSubject(s, s', p, j)
                        }
                  }
            }
      }
}


// Force at least one new notification to occur during a state transition.

fun Rule_ForceAtLeastOneNewNotification (s, s': State) {
      some o: s'.observers | {
            #SeqInds(s'.notifTrace[o]) = #SeqInds(s.notifTrace[o]) + 1
      }
}


// Generate a sequence of ordered states using valid state transitions.

fun generateTransitions () {
      InitTracesToEmpty(Ord[State].first)
      all disj s, s': State | {
            (s' = OrdNext(s)) implies {
                  Rule_NoTopologyChange(s, s')
                  Rule_AtMostOneNewNotifPerObserver(s, s')
            }
      }
}


// Generate a sequence of ordered states using valid state transitions.
// Ensure that at least one new notification occurs between states.

fun generateForcedTransitions () {
      InitTracesToEmpty(Ord[State].first)
      all disj s, s': State | {
            (s' = OrdNext(s)) implies {
                  Rule_NoTopologyChange(s, s')
                  Rule_AtMostOneNewNotifPerObserver(s, s')
                  Rule_ForceAtLeastOneNewNotification(s, s')
            }
      }
}
```

```
// Helper function that checks whether Observer "o" has received a new
// notification from Subject "j" during the state transition.

fun receivedNotifFromSubject (s, s': State, o: Observer, j: Subject) {
      s'.notifTrace[o] = SeqAdd(s.notifTrace[o], j)
}




//
// CONSISTENCY DEFINITIONS
//


// Ensures that two Observers are completely consistent, i.e. have been
// notified the same number of times and in the same relative order by
// the set of subjects being monitored by both.
//
// Example:
//
//    o.subjects = {S1, S2, S3}
//    p.subjects = {S1, S2, S4, S5}
//    o.notifTrace = {S1, S3, S2, S1, S3}
//    p.notifTrace = {S4, S1, S2, S5, S1}
//
// Here are two examples that are not completely consistent:
//
//    o.subjects = {S1, S2, S3}
//    p.subjects = {S1, S2, S4, S5}
//    o.notifTrace = {S1, S3, S2, S1, S3}
//    p.notifTrace = {S4, S1, S2, S5}           -- missing final S1
//
//    o.subjects = {S1, S2, S3, S5}
//    p.subjects = {S1, S2, S4, S5}
//    o.notifTrace = {S1, S3, S2, S1, S3} -- missing final S5
//    p.notifTrace = {S4, S1, S2, S1, S5}

fun CompletelyConsistent (s: State, o, p: Observer) {
      let commonSubjects = s.obsToSubMap[o] & s.obsToSubMap[p] | {
            sameRelativeSeqOrder(s.notifTrace[o], s.notifTrace[p], commonSubjects)
            &&
            sameRelativeSeqOrder(s.notifTrace[p], s.notifTrace[o], commonSubjects)
      }
}


// Ensures that two Observers are partially consistent, i.e. have been
// notified in the same relative order (but not necessarily the same
// number of times) by the set of subjects being monitored by both.
//
// Examples:
//
//    o.subjects = {S1, S2, S3}
//    p.subjects = {S1, S2, S4, S5}
//    o.notifTrace = {S1, S3, S2, S1, S3}
//    p.notifTrace = {S4, S1, S2, S5}
//
```

```
//    o.subjects = {S1, S2, S3, S5}
//    p.subjects = {S1, S2, S4, S5}
//    o.notifTrace = {S1, S3, S2, S1, S3}
//    p.notifTrace = {S4, S1, S2, S1, S5}
//
// Here is an example that is not partially consistent:
//
//    o.subjects = {S1, S2, S3}
//    p.subjects = {S1, S2, S4, S5}
//    o.notifTrace = {S1, S3, S2, S1, S3}
//    p.notifTrace = {S4, S2, S1, S1}           -- S2 and S1 out of order

fun PartiallyConsistent (s: State, o, p: Observer) {
      let commonSubjects = s.obsToSubMap[o] & s.obsToSubMap[p] | {
            sameRelativeSeqOrder(s.notifTrace[o], s.notifTrace[p], commonSubjects)
            ||
            sameRelativeSeqOrder(s.notifTrace[p], s.notifTrace[o], commonSubjects)
      }
}


// Ensures that sequence A has the same relative ordering and at most the same
// number of elements as sequence B with respect to the elements in "elementList".
// Note that this function does NOT guarantee that the same holds true for
// sequence B with respect to sequence A.
//
// Example:
//
//    elementList = {S1, S2}
//    seqA = {S4, S1, S2, S5}
//    seqB = {S1, S3, S2, S1, S3}
//
// Here are 2 examples that do not satisfy this function:
//
//    elementList = {S1, S2}
//    seqA = {S1, S3, S2, S1, S3}         -- has extra S1
//    seqB = {S4, S1, S2, S5}
//
//    elementList = {S1, S2}
//    seqA = {S1, S2, S1}                 -- missing an S2
//    seqB = {S1, S2, S2, S1}

fun sameRelativeSeqOrder[t] (seqA, seqB: Seq[t], elementList: set t) {

      --
      -- If seqA has the same relative ordering and at most the same
      -- number of elements as sequence B (with respect to those
      -- elements in "elementList"), then the following
      -- invariant holds:
      --
      --    For all indicies i in seqA,
      --          There is some index j in seqB, such that
      --                The number of each element in "elementList" up to index i
      --                in seqA equals the number of the same element up to index
j
      --                in seqB.
      --
      --
      -- We simply translate the above invariant into Alloy and assume it holds
```

```
	-- for the input sequences.
	--

	// If elementList is empty, then the invariant is vacuously true.
	no elementList ||

	// If seqA doesn't contain any elements in "elementList", then the
	// invariant is vacuously true (handles empty seqA case).
	no (SeqElems(seqA) & elementList) ||

	// Otherwise, seqB can't be empty, and it must contain at least one
	// element in "elementList" for the invariant to possibly hold.
	(not SeqIsEmpty(seqB) &&
	some (SeqElems(seqB) & elementList) &&
	all i: SeqInds(seqA) | {
		some j: SeqInds(seqB) | {
			all elem: elementList | {
				#(elem.~(((OrdPrevs(i) + i) -> t) & seqA.seqElems)) =
				  #(elem.~(((OrdPrevs(j) + j) -> t) & seqB.seqElems))
			}
		}
	})
}




//
// TOPOLOGY FUNCTIONS
//

// Creates a topology that only allows observers to view
// a single subject.
//
// This topology will be known as the "MostRestrictiveTopology".

fun Init_MostRestrictiveTopology (s: State) {
	all o: s.observers | {
		#(s.obsToSubMap[o]) < 2
	}
}


// Creates a topology that allows multiple observers to view
// the same set of subjects--that is, the subjects in a State
// are partitioned into one or more distinct sets, and each
// Observer views subjects in at most one of these sets.
//
// This topology will be known as the "SemiRestrictiveTopology".

fun Init_SemiRestrictiveTopology (s: State) {
	-- For any two Observers in the State, the sets of subjects being
	-- viewed by each either do not overlap or completely overlap.
	all o, p: s.observers | {
		no (s.obsToSubMap[o] & s.obsToSubMap[p]) ||
		s.obsToSubMap[o] = s.obsToSubMap[p]
	}
}
```

```
// Creates a topology that allows observers to view arbitrary
// sets of subjects, independent of other observers.
//
// This topology will be known as the "LeastRestrictiveTopology".

fun Init_LeastRestrictiveTopology (s: State) {
      -- Nothing to do!  This is the default behavior of the model.
}


// Force some kind of mapping (topology) between Subjects and Observers.

fun ForceSomeTopology (s: State) {
      some s.observers
      some s.subjects
      some s.subToObsMap
}


// Force the existence of multiple subjects and observers in the state.

fun ForceMultipleSubjectsAndObservers (s: State) {
      #s.observers > 1
      #s.subjects > 1
}


// Force at least two subjects to be viewed.

fun ForceAtLeastTwoSubjectsToBeViewed (s: State) {
      #s.obsToSubMap[Observer] > 1
}


// Force at least one Subject to have at least two observers.

fun ForceAtLeastOneSubjectToHaveAtLeastTwoObservers (s: State) {
      some j: Subject | #s.subToObsMap[j] > 1
}


// Force at least one Observer to view at least two subjects (note:
// this cannot be used with the MostRestrictiveTopology).

fun ForceAtLeastOneObserverToViewAtLeastTwoSubjects (s: State) {
      some o: Observer | #s.obsToSubMap[o] > 1
}


// Force at least two Observers to view at least two subjects (note:
// this cannot be used with the MostRestrictiveTopology).

fun ForceAtLeastTwoObserversToViewAtLeastTwoSubjects (s: State) {
      some disj o, p: Observer | {
            #s.obsToSubMap[o] > 1
            #s.obsToSubMap[p] > 1
      }
}
```

```
// Force at least two observers to not view all the same subjects.

fun ForceAtLeastTwoObserversToHaveDifferentSubjectSets (s: State) {
      some disj o, p: s.observers | {
            s.obsToSubMap[o] != s.obsToSubMap[p]
      }
}


// Force at least two observers to have partially (but not completely)
// overlapping sets of subjects (note:  this only works for the
// LeastRestrictiveTopology).

fun ForceAtLeastTwoPartiallyOverlappingSubjectSets (s: State) {
      some disj o, p: s.observers | {
            some (s.obsToSubMap[o] & s.obsToSubMap[p])
            s.obsToSubMap[o] != s.obsToSubMap[p]
      }
}




//
// CONSISTENCY INVARIANTS & ASSERTIONS
//


// INVARIANT:  for all possible sequences of states, every
// pair of Observers is always partially consistent.

fun AlwaysPartiallyConsistentInvariant () {
      all s: State | {
            all o, p: s.observers | {
                  PartiallyConsistent(s, o, p)
            }
      }
}


// Assert the AlwaysPartiallyConsistentInvariant holds for the MostRestrictive
// topology.
//
// Note:  this assertion holds.

assert AlwaysPartiallyConsistent_MostRestrictiveTopology {
      {Init_MostRestrictiveTopology(Ord[State].first) && generateTransitions()}

      implies

      AlwaysPartiallyConsistentInvariant()
}


// Assert the AlwaysPartiallyConsistentInvariant holds for the SemiRestrictive
// topology.
//
```

```
// Note:  this assertion holds.

assert AlwaysPartiallyConsistent_SemiRestrictiveTopology {
	{
		Init_SemiRestrictiveTopology(Ord[State].first)
		generateTransitions()

		-- Additional transition rules required for assertion to hold.
		all disj s, s': State | {
			(s' = OrdNext(s)) implies {
				Rule_NotifyOneNotifyAllInSameTransition(s, s')
			}
		}
	}

	implies

	AlwaysPartiallyConsistentInvariant()
}


// Assert the AlwaysPartiallyConsistentInvariant holds for the LeastRestrictive
// topology.
//
// Note:  this assertion holds.

assert AlwaysPartiallyConsistent_LeastRestrictiveTopology {
	{
		Init_LeastRestrictiveTopology(Ord[State].first)
		generateTransitions()

		-- Additional transition rules required for assertion to hold.
		all disj s, s': State | {
			(s' = OrdNext(s)) implies {
				Rule_NotifyOneNotifyAllInSameTransition(s, s')
			}
		}
	}

	implies

	AlwaysPartiallyConsistentInvariant()
}



// INVARIANT:  for all possible sequences of states, every
// pair of Observers is always completely consistent.

fun AlwaysCompletelyConsistentInvariant () {
	all s: State | {
		all o, p: s.observers | {
			CompletelyConsistent(s, o, p)
		}
	}
}


// Assert the AlwaysCompletelyConsistentInvariant holds for the MostRestrictive
// topology.
//
// Note:  this assertion holds.

assert AlwaysCompletelyConsistent_MostRestrictiveTopology {
      {
            Init_MostRestrictiveTopology(Ord[State].first)
            generateTransitions()

            -- Additional transition rules required for assertion to hold.
            all disj s, s': State | {
                  (s' = OrdNext(s)) implies {
                        Rule_NotifyOneNotifyAllInSameTransition(s, s')
                  }
            }
      }

      implies

      AlwaysCompletelyConsistentInvariant()
}


// Assert the AlwaysCompletelyConsistentInvariant holds for the SemiRestrictive
// topology.
//
// Note:  Alloy seems to crash on this one.

assert AlwaysCompletelyConsistent_SemiRestrictiveTopology {
      {
            Init_SemiRestrictiveTopology(Ord[State].first)
            generateTransitions()

            -- Additional transition rules required for assertion to hold.
            all disj s, s': State | {
                  (s' = OrdNext(s)) implies {
                        Rule_NotifyOneNotifyAllInSameTransition(s, s')
                  }
            }
      }

      implies

      AlwaysCompletelyConsistentInvariant()
}


// Assert the AlwaysCompletelyConsistentInvariant holds for the LeastRestrictive
// topology.
//
// Note:  Alloy seems to crash on this one.

assert AlwaysCompletelyConsistent_LeastRestrictiveTopology {
      {Init_LeastRestrictiveTopology(Ord[State].first) && generateTransitions()}

      implies

      AlwaysCompletelyConsistentInvariant()
}
```

```
//
// MODEL DEBUGGING ASSERTIONS
//


// If two Observers are completely consistent, it should hold by definition that
// they are partially consistent as well.

assert CompleteImpliesPartialConsistency {
      all s: State | {
            all o, p: s.observers | CompletelyConsistent(s, o, p) implies
                  PartiallyConsistent(s, o, p)
      }
}


// This assertion is FALSE.  Partial consistency does not guarantee complete
// consistency.

assert FALSE_PartialImpliesCompleteConsistency {
      all s: State | {
            all o, p: s.observers | PartiallyConsistent(s, o, p) implies
                  CompletelyConsistent(s, o, p)
      }
}


// The order of arguments to CompletelyConsistent and PartiallyConsistent does not
// matter.

assert ArgumentOrderNotImportant {
      all s: State | {
            all o, p: s.observers | {
                  CompletelyConsistent(s, o, p) implies
                        CompletelyConsistent(s, p, o)
                  PartiallyConsistent(s, o, p) implies
                        PartiallyConsistent(s, p, o)
            }
      }
}


// Two Observers whose notifTraces are both empty are always completely and
// partially consistent.

assert EmptyTraceConsistency {
      all s: State | {
            all o, p: s.observers | {
                  ((SeqIsEmpty(s.notifTrace[o])) && (SeqIsEmpty(s.notifTrace[p])))
                    implies {
                        CompletelyConsistent(s, o, p)
                        PartiallyConsistent(s, o, p)
                  }
            }
      }
```

```
}


// Two Observers where only one observer has an empty notifTrace are always
// partially consistent.

assert OneEmptyTraceImpliesPartialConsistency {
      all s: State | {
            all o, p: s.observers | {
                  ((SeqIsEmpty(s.notifTrace[o])) &&
                    (not SeqIsEmpty(s.notifTrace[p]))) implies {
                        PartiallyConsistent(s, o, p)
                  }
            }
      }
}


// This assertion is FALSE.  Two Observers where only one observer has an empty
// notifTrace does not imply that the observers are always completely consistent.

assert FALSE_OneEmptyTraceImpliesCompleteConsistency {
      all s: State | {
            all o, p: s.observers | {
                  ((SeqIsEmpty(s.notifTrace[o])) &&
                    (not SeqIsEmpty(s.notifTrace[p]))) implies {
                        CompletelyConsistent(s, o, p)
                  }
            }
      }
}


// Two Observers who are not monitoring common subjects are always both completely
// and partially consistent.

assert NoCommonSubjectsConsistency {
      all s: State | {
            all o, p: s.observers | {
                  let commonSubjects = s.obsToSubMap[o] & s.obsToSubMap[p] | {
                        no commonSubjects implies {
                              CompletelyConsistent(s, o, p)
                              PartiallyConsistent(s, o, p)
                        }
                  }
            }
      }
}


// Asserts that, by definition, all observers in the system have a
// notification trace.

assert AllObserversHaveTrace {
      all s: State | {
            some s.observers implies {
                  all o: s.observers | some s.notifTrace[o]
            }
      }
```

```
}




//
// EXAMPLE TRACE GENERATION FUNCTIONS
//


// Generate an interesting state trace of the MostRestrictive topology.

fun generateMostRestrictiveTopologyTrace () {
      Init_MostRestrictiveTopology(Ord[State].first)

      ForceSomeTopology(Ord[State].first)
      ForceMultipleSubjectsAndObservers(Ord[State].first)
      ForceAtLeastTwoSubjectsToBeViewed(Ord[State].first)
      ForceAtLeastTwoObserversToHaveDifferentSubjectSets(Ord[State].first)
      ForceAtLeastOneSubjectToHaveAtLeastTwoObservers(Ord[State].first)

      generateForcedTransitions()
}


// Generate an interesting state trace of the SemiRestrictive topology.

fun generateSemiRestrictiveTopologyTrace () {
      Init_SemiRestrictiveTopology(Ord[State].first)

      ForceSomeTopology(Ord[State].first)
      ForceMultipleSubjectsAndObservers(Ord[State].first)
      ForceAtLeastTwoSubjectsToBeViewed(Ord[State].first)
      ForceAtLeastTwoObserversToHaveDifferentSubjectSets(Ord[State].first)
      ForceAtLeastOneSubjectToHaveAtLeastTwoObservers(Ord[State].first)

      ForceAtLeastOneObserverToViewAtLeastTwoSubjects(Ord[State].first)
      ForceAtLeastTwoObserversToViewAtLeastTwoSubjects(Ord[State].first)

      generateForcedTransitions()

      -- Additional transition rules.
      all disj s, s': State | {
            (s' = OrdNext(s)) implies {
                  --Rule_NotifyOneNotifyAllInSameTransition(s, s')
            }
      }
}


// Generate an interesting state trace of the LeastRestrictive topology.

fun generateLeastRestrictiveTopologyTrace () {
      Init_LeastRestrictiveTopology(Ord[State].first)

      ForceSomeTopology(Ord[State].first)
      ForceMultipleSubjectsAndObservers(Ord[State].first)
      ForceAtLeastTwoSubjectsToBeViewed(Ord[State].first)
      ForceAtLeastTwoObserversToHaveDifferentSubjectSets(Ord[State].first)
      ForceAtLeastOneSubjectToHaveAtLeastTwoObservers(Ord[State].first)

      ForceAtLeastOneObserverToViewAtLeastTwoSubjects(Ord[State].first)
      --ForceAtLeastTwoObserversToViewAtLeastTwoSubjects(Ord[State].first)
      ForceAtLeastTwoPartiallyOverlappingSubjectSets(Ord[State].first)

      generateForcedTransitions()
}





//
// COMMANDS
//


// Consistency Assertions:
check AlwaysPartiallyConsistent_MostRestrictiveTopology for 4
check AlwaysPartiallyConsistent_SemiRestrictiveTopology for 4
check AlwaysPartiallyConsistent_LeastRestrictiveTopology for 4

check AlwaysCompletelyConsistent_MostRestrictiveTopology for 4
check AlwaysCompletelyConsistent_SemiRestrictiveTopology for 4
check AlwaysCompletelyConsistent_LeastRestrictiveTopology for 4



// Generate Example Traces:
run generateMostRestrictiveTopologyTrace for 3
run generateSemiRestrictiveTopologyTrace for 3
run generateLeastRestrictiveTopologyTrace for 3



// Debugging Assertions (sanity checks):
-- No counterexamples should exist.
--check CompleteImpliesPartialConsistency for 4
--check ArgumentOrderNotImportant for 4
--check EmptyTraceConsistency for 4
--check OneEmptyTraceImpliesPartialConsistency for 4
--check NoCommonSubjectsConsistency for 4
--check AllObserversHaveTrace for 4
-- A counterexample should exist.
--check FALSE_PartialImpliesCompleteConsistency for 4
--check FALSE_OneEmptyTraceImpliesCompleteConsistency for 4
```