Case Study in Alloy Modeling: A Common Profile for Presence

Edmond Lau

Computer Science and Artificial Intelligence Laboratory: Software Design Group
Massachusetts Institute of Technology

Abstract

Using Alloy as a software modeling tool, I conducted a case study of a presence protocol by the Instant Messaging and Presence Protocol working group. I questioned major design choices and uncovered inconsistencies in the Common Profile for Presence protocol – a protocol that enables a client to subscribe to other clients and subsequently receive notifications regarding changes in the presence information of those clients.

In this paper, I present the results of my modeling experience. Both the formal analysis of the final model and the rigorous thinking that accompanied the construction of the model evoked a reconsideration of areas of interest; these areas of interest included protocol operations, subscription management, multi-located clients, privacy concerns, and asynchronous notifications.

My purpose in conducting this case study is to aid the working group in its development of the instant messaging and presence protocols and to demonstrate the utility of abstract modeling in the early stages of software design. In the absence of this modeling, the issues described in this paper might have propagated into the implementation phase of the protocol, thereby wasting future time and effort investments. Software modeling thus serves as an invaluable thinking aid during design.

1 Introduction

The Instant Messaging and Presence Protocol (IMPP) [1] working group within the Internet Engineering Task Force seeks to develop a common architectural standard for web-based systems of presence awareness, presence notification, and instant messaging. In particular, the group aims to facilitate the creation of common channels among presence services that allow clients to subscribe to each other and receive notifications regarding future changes of state. The common profile for presence (CPP) functions as a protocol through which clients can subscribe and unsubscribe to a presence service that sends out notifications regarding changes in the presence information of other clients.

IMPP has formalized an architecture for presence awareness and notification. Like any other software system or protocol, the design of the CPP architecture is perhaps the optimal phase of design and development in which to increase confidence and verify the soundness of the protocol. Software systems have lived or died by their design, and building micromodels of crucial design elements can often tilt the scales toward a stronger software system.

To demonstrate the feasibility and the benefits of software modeling during a system's design phase, I have modeled CPP using the Alloy language and the Alloy Analyzer and documented in this paper the important issues that the modeling experience has revealed. In this paper, I present the results of my Alloy modeling experience. Both the analysis of the model itself and the thinking that accompanied the experience of articulating a model have helped to offer clarity and insight into the design of CPP; both have shed light upon unconsidered design issues, inconsistencies, and ambiguities in the CPP protocol. The purpose of this paper lies not so much in answering all the questions regarding the protocol, but in discovering which questions remain yet to be answered.

1.1 Software Modeling as a Design Aid

The engineer's approach to solving a real-life problem involves reducing the actual, complicated system that he is faced with into a smaller, simpler, more tractable system that still retains the salient features. Electrical engineers approximate non-linear circuit elements using linear equations; system engineers build diagrams of communications and control systems; mechanical engineers derive mathematical equations to describe physical phenomena. The central theme among all of these approaches rests in the judicious use of models to make difficult problems solvable. And yet often times, software engineers are left behind blindly hacking at code, making surprisingly little use of this reductionist approach in software design.

Faced with a design problem, models need not completely describe the actual system to be useful; in fact, by focusing only on the relevant concepts and relations, one can still greatly reduce the complexity of thousands of lines of code and specifications into more manageable models. Such modeling increases confidence regarding the soundness of software design and reveals bugs early so that less time and effort become wasted project investments. Thus, software modeling functions as a valuable aid to analyzing design problems in the initial stages of software development and also to checking the robustness of currently existing software.

[1] The homepage for Instant Messaging and Presence Protocol Working Group can be found at http://www.ietf.org/html.charters/impp-charter.html.

1.2 Background on the Alloy Analyzer

The Alloy Analyzer 3.0 is a tool developed by the Software Design Group at the MIT Computer Science and Artificial Intelligence Laboratory to analyze models written in Alloy – a simple declarative language used to describe first-order relations between different elements. The formal, mathematical nature of the Alloy language provides a mechanism for the user to articulate the structure and the properties that characterize a system.

The beauty of this language is that the associated Alloy Analyzer can simulate instances of the described model and check for counterexamples against certain properties and invariants assumed to hold true about the system. The tool can simulate operation sequences, generate examples based on system specifications, and check properties specified by the user. The issues described in this paper were all raised during an Alloy modeling experience; moreover, all accompanying visualizations were generated using the Alloy Analyzer 3.0 visualizer.

More information regarding Alloy can be found at http://alloy.mit.edu/.

2 Common Profile for Presence Protocol

The Alloy model for the Common Profile for Presence draws from the following two documents on the IMPP charter webpage: "A Model for Presence and Instant Messaging" (RFC2778) [2] and "A Common Profile for Presence" (CPP).[3] RFC2778 defines an abstract model and language for talking about instant messaging and presence systems; the CPP document describes a possible operational sequence and discusses implementation issues regarding a possible construction of a presence awareness and notification system.

In this section, I summarize the communication model described in RFC2778 and the protocol operations described in the CPP document. The analysis in the rest of this paper primarily uses the CPP document to generate unconsidered design issues and ambiguities, appealing to RFC2778 only to point out inconsistencies between the two papers.

2.1 Abstract Model of the Presence Service

In the model defined by RFC2778, a *presence service* functions as the server application that accepts, stores, and distributes the presence information, i.e. location, of client applications. It serves two types of clients: presentities and watchers. *Presentities* consist of those clients that communicate their presence information to the service for storage and distribution; *watchers* comprise those that request and receive presence information from the service. Watchers fall into two different categories: *fetchers*, who request the current presence information of a particular presentity from the presence service, and *subscribers*, who request notifications for future changes in a presentity's presence information.

2.2 Protocol Operations

The CPP document describes the basic operations that comprise the presence protocol. Watchers invoke a *subscribe* operation for to request presence information to receive notifications regarding future changes of a target presentity specified in the subscribe message. Upon receiving the message, the presence service invokes a *response* operation to indicate whether the subscription succeeded or failed. If the subscription was successful, the presence service invokes the *notify* operation to send the target presentity's presence information to the watcher.

The *subscribe* operation also includes a duration field; the presence service continues to notify the subscriber of any future changes of state. A successful subscribe operation with a zero duration functions as both a *fetch* operation and also as an *unsubscribe* operation; in either case, the presence service will notify the watcher of the presentity's information. In Section 4, I argue that this dual meaning causes several design problems.

In a typical interaction, a watcher subscribes to a presence service to receive notifications regarding the presence information of a presentity. If the subscription is successful, the presence service notifies the watcher of the presentity's location and continues to send notifications whenever the presentity's information changes.

3 The Alloy Model

This section outlines the basic signatures, predicates, and patterns used in the Alloy model. Moreover, it explains the correspondence between the Alloy elements and the elements of the presence model described in the previous section and examines the differences between the two models. The complete cpp model is provided in the appendix.

3.1 Alloy Signatures

The Alloy model consists of the following basic signatures, which can be considered to be sets:

```
sig Presentity {}
sig Watcher {}

sig Location {}
sig Hidden extends Location {}

sig Subscription {
  owner: Watcher,
  target: Presentity
}
```

[2] RFC2778 can be found at http://www.ietf.org/rfc/rfc2778.txt.

[3] "A Common Profile for Presence" can be found at http://www.ietf.org/internet-drafts/draft-ietf-impp-pres-04.txt.

```
sig Notification {
  receiver: Watcher,
  target: Presentity,
  loc: Location
}

sig State {
  presenceInfo: Presentity ->+ Location,
  activeSubs: set Subscription,
  message: option Notification
}
```

Most of the signatures follow straightforwardly from the presence model. The `Presentity` and the `Watcher` correspond to the two types of client applications of the presence service. A `Location` represents the value of presence information; `Hidden` denotes the location of a presentity that has configured his access rules so as to disallow watchers from retrieving his presence information. A `Subscription` represents the result of a successful subscribe operation; it includes `owner` and `target` relations that specify the watcher who requested the subscription and the target presentity whose presence information is desired, respectively. A `Notification` corresponds to the message sent by the notify operation; the presence service sends the message to a receiver regarding a change in a `target` presentity's presence information to the new value `loc`.

A `State` is not explicitly described in the presence model; it represents a snapshot of the presence service's state at a particular point in time. A sequence of ordered states thus specifies a particular operation sequence of interactions between the presence service and its clients. The `presenceInfo` relation maps each presentity to its location(s) at the time of the particular state. The `activeSubs` relation describes the set of active subscriptions; this set corresponds to a snapshot of the subscription list maintained by the presence service. Finally, the `message` relation optionally identifies a notification that is sent by the presence service at that time.

3.2 Protocol Operations

The Alloy model implements the following fundamental operations of the CPP protocol as predicates:

```
pred move(s, s': State, p: Presentity,
          l: Location) {
  let pres = s.presenceInfo,
    pres' = s'.presenceInfo {
      l not in Hidden
      p->l not in pres
      let oldLoc = p.pres |
        pres' = pres - p->oldLoc + p->l
  }
}

pred hide(s, s': State, p: Presentity) {
  let pres = s.presenceInfo,
    pres' = s'.presenceInfo {
      some Hidden
      p in pres.(Location - Hidden)
      some oldLoc: p.pres |
        pres' = pres - p->oldLoc + p->Hidden
  }
}

pred subscribe(s, s': State, w: Watcher,
               p: Presentity) {
  some sub: Subscription + s.activeSubs {
    sub.owner = w
    sub.target = p
    s'.activeSubs = s.activeSubs + sub
  }
}

pred unsubscribe(s, s': State, w: Watcher,
                 p: Presentity) {
  some sub: s.activeSubs {
    sub.owner = w
    sub.target = p
    s'.activeSubs = s.activeSubs - sub
  }
}

pred notify(s: State, w: Watcher,
            p: Presentity, l: Location) {
  some n: Notification {
    n = s.message
    n.receiver = w
    n.target = p
    n.loc = l
  }
}
```

The `move` and `hide` operations allow a presentity to change his current location from one state to the next, either by updating its presence information to a new location or to `Hidden`, respectively. The Alloy model assumes that any location changes are immediately reflected in the presence information of the presence service. The `subscribe` operation corresponds to the one from the presence model and adds a subscription to the presence service's subscription list. For simplicity, all `subscribe` operations are assumed to succeed and no response operations are required. Rather than modeling the `unsubscribe` operation as subscribing with a zero duration, the model abstracts away the implementation detail and explicitly models an `unsubscribe` operation. Finally, the `notify` operation communicates a notification from the presence service to a watcher with a presentity's updated location.

Figure 1 shows a visualization of a basic two-state operation sequence. In `State_0`, the presence server contains no presence information. In `State_1`, `Watcher_0` invokes a subscribe operation on `Presentity_0`, thereby creating a subscription in the state's `activeSubs`. As required by the protocol, a notify operation occurs in response to the subscription, and `State_2` shows `Notification_0` being sent to `Watcher_0` with `Presentity_0`'s location.

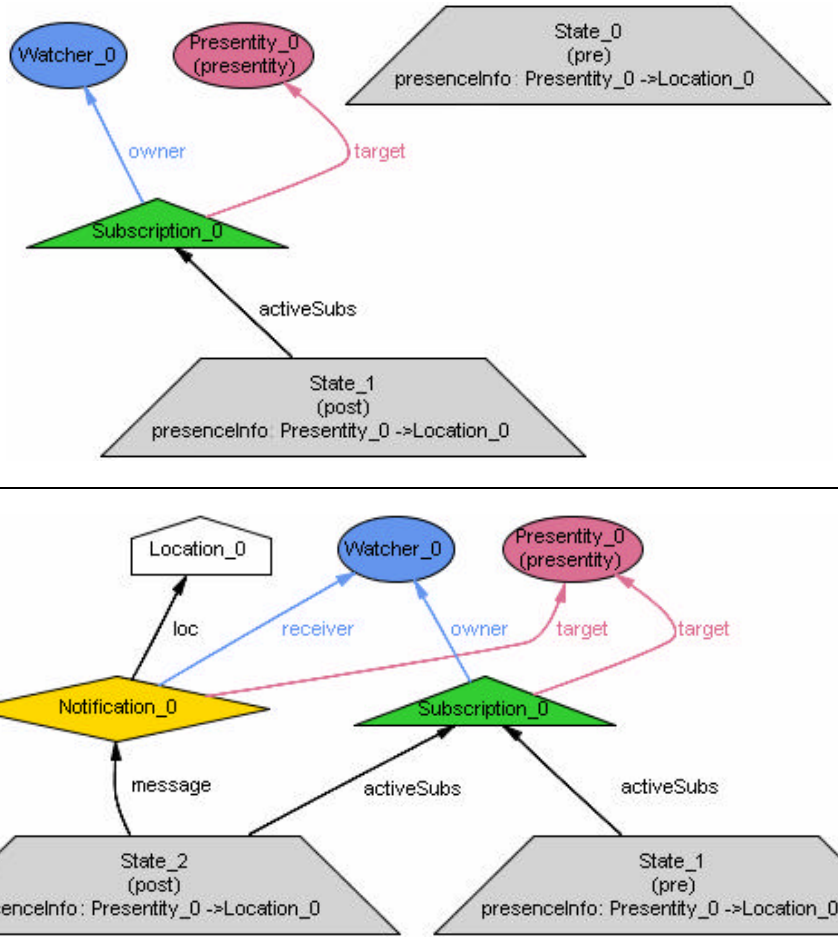

Figure 1: A typical subscribe and notify operation sequence in CPP involves a SubscribeEvent (top visualization) followed by a NotifyEvent (bottom visualization).

3.3 Event Modeling for Improved Visualizations

The Alloy Analyzer visualizes signatures and relations but not predicates; thus, visualizations based on a model with the aforementioned signatures and predicates fail to illustrate which operation occurs between each state. In order to visualize the operations, I parallel each operation with an Event signature:

```
abstract sig Event { pre, post: State,
   presentity: Presentity }
sig MoveEvent extends Event { newLoc: Location }
sig HideEvent extends Event {}
sig SubscribeEvent extends Event {
  subscriber: Watcher}
sig UnsubscribeEvent extends Event {
  unsubscriber: Watcher}
sig NotifyEvent extends Event { recipient: Watcher,
   changedLoc: Location }
```

Then, by specifying the transitions between states and by constraining only one event to occur between each pair of states, I was able to perform projections onto Event and easily visualize operation sequences like the one in Figure 1.

The following Alloy snippet illustrates this modeling technique:

```
pred trans(s, s': State) {
  some e: Event {
    e.pre = s
    e.post = s'

    (e in MoveEvent &&
      move(e.pre, e.post, e.presentity, e.newLoc) &&
      updateSubscriptions(e.pre, e.post)) ||
    (e in HideEvent &&
      hide(e.pre, e.post, e.presentity) &&
      updateSubscriptions(e.pre, e.post)) ||
    (e in SubscribeEvent &&
      subscribe(e.pre, e.post, e.subscriber,
        e.presentity) &&
      samePresenceInfo(e.pre, e.post)) ||
    (e in UnsubscribeEvent &&
      unsubscribe(e.pre, e.post, e.unsubscriber,
        e.presentity) &&
      samePresenceInfo(e.pre, e.post)) ||
    (e in NotifyEvent &&
                    ...
```

3.4 Differences between CPP and Alloy Model

For the most part, I have strived to maintain the integrity of the Alloy model and its correspondence with the CPP description. Nonetheless, a number of differences were judicially chosen to keep the model simple:

- TransIDs, which are nonces used to correlate subscribe and response operations, and SubscriptionIDs, which are used to reference an existing subscription when unsubscribing, have been ignored in the model as low-level details.
- The response operation that follows each subscribe operation has also been omitted from the model because it carried only a status flag and communicated no presence information.
- To represent the manipulation of access privileges to hide presence information, a presentity moves into a Hidden location. Besides stating that certain configurations of access privileges may cause subscriptions to fail, RFC2778 and the CPP document have both been silent on how the presence information is represented.
- Subscription duration values been omitted for simplicity.

In addition to these differences, I have also made two simplifying assumptions. First, the presence service is considered to be a centralized server that handles requests and responses; all changes in presence and location information in the server are implicitly modeled through the transition sequence of the server's states. Second, I have focused primarily on the presentities' locations, but the analysis for watchers' locations would be similar.

4 Protocol Design Issues Identified

This section formulates the findings and the analysis that ensued from modeling the CPP protocol; the results are formulated as design critiques categorized into several overarching realms, with accompanying visualizations generated by the Alloy Analyzer 3.0 visualizer. The analysis uses the more concrete CPP document to introduce unconsidered design issues and ambiguities, citing appropriate sections of the protocol as necessary. It also appeals to RFC2778 to illuminate apparent inconsistencies that exist between the two papers. In considering the issues identified, the important questions to ask include: Have these design issues been considered? If so, what are the answers? If not, the protocol may need to be revised to take these issues into consideration.

4.1 Subscribe vs. Unsubscribe Operations, Fetcher vs. Subscriber

A number of shortcomings and inconsistencies arise in the CPP protocol's usage of a single subscribe operation to handle fetching presence information, subscribing to the presence information of a presentity, and unsubscribing from that information. The CPP protocol essentially overloads the subscribe operation with special and seemingly incongruent interpretations when the duration is zero, hinting to a questionable design choice. Pages 7-8 of the CPP describe the dual meanings behind the message subscribe 0:

> "When an application wants to terminate a subscription, it issues a SUBSCRIBE 0 with the SubscriptID of an existing subscription. Note that a notify operation will be invoked by the presentity when a subscription is canceled in this fashion; this notification can be discarded by the watcher. There is no independent UNSUBSCRIBE operation.
>
> When an application wants to directly request presence information to be supplied immediately without initiating any persistent subscription, it issues a SUBSCRIBE 0 with a new SubscriptID. There is no independent FETCH operation."

The difference between an unsubscribe and a fetch operation appears to be whether the SubscriptID refers to an existing subscription or is a new one. The first implication of this design is the requirement of client-side state; each watcher must maintain the SubscriptIDs of all of its existing subscriptions. Given the simplicity of the CPP protocol, the implementation detail begs the question of whether such a requirement is indeed necessary; separate unsubscribe and fetch operations would easily eliminate the client's burden of maintaining state information.

In my attempt to articulate the subscribe and unsubscribe operations in my Alloy model, I realized that the overloading of the subscribe operation posed a number of major issues, including:

1. What is the design reasoning behind not having separate unsubscribe and/or fetch operations?
2. What is the justification for the requirement of client-side state in subscriptions?
3. Why, as illustrated in Figure 2, does a watcher receive more presence information when he has explicitly expressed his desire not to receive any more such information via an unsubscribe operation?
4. If a watcher is already subscribed to a particular presentity, its fetch operation on that presentity will look the same as an unsubscribe operation. Why can a subscriber not send a one-time request for current presence information of a target without terminating his subscription of that target?

The last of the above questions carries another repercussion. The implementation detail of utilizing subscribe 0 precludes a subscriber from simultaneously acting as a fetcher for the same target. The mutual exclusivity of subscribers and fetchers as subclasses of watchers is inconsistent with the specification developed in RFC2778, which makes no indication that the two types of

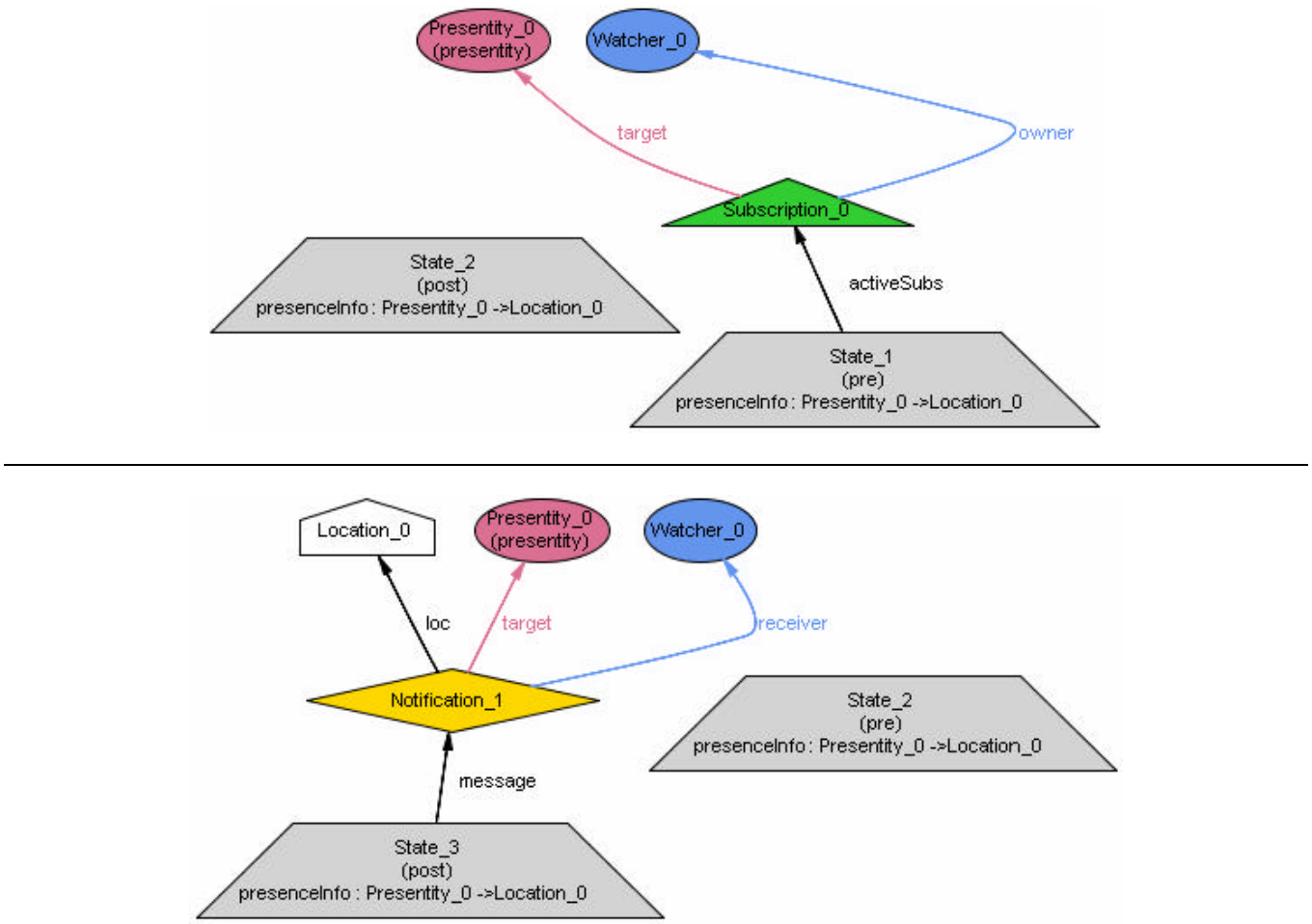

Figure 2: An unsubscribe operation (top visualization) causes an undesired notification (bottom visualization).

watchers (fetchers and subscribers) must be disjoint groups for a given target. On a behavioral level, fetchers and subscribers perform fundamentally different tasks. Whereas a fetcher requests the *current* value of a presentity's presence information, a subscriber requests notifications of *future* changes in some presentity's presence information. On a more practical level, if a subscriber loses the most recent notification from the presence service, he has no mechanism of retrieving the current presence information of the target presentity short of unsubscribing.

4.2 Subscription Durations

A subscribe operation carries with it a watcher-specified duration attribute specifying the desired number of seconds that the subscription should be active. In carefully perusing the CPP document and deciding whether to include this notion of subscription durations and also the related response operations in the Alloy model, I came upon a number of issues.

First, the description of the response operation, which also contains its own duration attribute, offers no guarantee that the subscribe operation's requested duration is followed:

> "The response operation has the following attributes: status, TransID, and duration…The 'duration' attribute specifies the number of seconds for which the subscription will be active (which may differ from the value requested in the subscribe operation)."

The specification of subscription durations is underspecified; the CPP protocol leaves unanswered how exactly the duration of a subscription is determined. The lack of a guiding principle or guarantee in the determination of subscription durations raises a number of issues that ought to be explicitly stated in the protocol specification:

1. If the response's duration can be different from duration in the corresponding subscribe operation, which is followed?

2. If the response's duration is followed, what is the purpose of the subscribe operation's original "duration" time?
3. Can the response actually subscribe a watcher for a longer period of time than the watcher intended under his original subscribe operation?

Second, the following rule described in CPP page 7, which rejects a watcher's subscribe operations if another subscription to the same target already exists, also elucidates additional limitations to the duration attribute's specification:

> "If the duration parameter is non-zero, and if the watcher and target parameters refer to an in-progress subscribe operation for the application, a response operation having status "failure" is invoked."

In particular, the above rule precludes the ability to extend or shorten subscriptions except in a roundabout way. Subscribers cannot merely resend a subscribe operation with a new duration operation that overwrites the pervious one; rather, they must first unsubscribe and then re-subscribe with the desired remaining time. Moreover, because only subscribe 0 passes unchecked through the rule, the notion of an incongruent design for subscriptions arises once more and lends itself again to the question of why not have separate unsubscribe and fetch operations instead of subscribe 0. Nonetheless, the rule motivates the following questions:

4. What does the duration attribute of a response operation carry when the status is "failed"?
5. Why can a subscriber not change his subscription duration by overwriting it and instead must first unsubscribe and then re-subscribe with the desired remaining time? What design benefit motivated the blanket rejection of non-zero duration subscriptions with the same receiver and target as a previous subscription?

4.3 Multi-located Presentities

The proposed PRES URI scheme as documented in Appendix A of the CPP protocol identifies clients with syntax that follows the existing "mailto: URI" syntax, i.e. "pres:edmond@mit.edu." However, presentities may be present at two or more locations at any given time; for example, users of AOL Instant Messenger or MIT's zephyr can in fact be logged into two different machines and thus be at two distinct locations.

Assuming that the presence service must maintain a mapping of each presentity to all of its locations, a number of considerations arise regarding the CPP protocol. From the perspective of the presentity, the following issues must be considered:

1. How many locations can a presentity be at once, and more generally, is a unique resource or identifier required for each location of a presentity?
2. Will a subscriber receive notifications from the presence service when the any of the presentity's locations undergoes a change?
3. If so, how can the watcher tell which location of the presentity changed if presence notifications only indicate the new location and not the old?
4. Can the same presentity at different locations have different levels of access control? Can subscribing to one particular location's presentity succeed while another fails?

The issue is further complicated by the fact that watchers must not only deal with these multi-located presentities, but may also themselves have multiple locations. Thus, the following issues must also be addressed:

5. How many locations can a watcher be at once?
6. Should the watcher be able to subscribe to a presentity only at a specified location if the presentity has multiple locations? Or must he subscribe to them all?
7. Will the watcher at each location receive a separate notification when a presentity to which he is subscribed moves?
8. Will subscribing at one location subscribe just the watcher at that location, or will it subscribe the watcher at all locations?
9. Will responses be sent to just the watcher at the location that interacted with the presence service or will they be sent to the watcher at all locations?

4.4 Privacy Concerns

RFC2778 defines a provision known as access rules that presentities can manipulate to constrain how much presence information the presence service divulges. CPP touches upon these privacy issues, by providing a "failure to subscribe" (CPP 7) response to be sent after subscription:

> "If access control does not permit the application to request this [subscribe] operation, a response operation having status "failure" is invoked."

However, the CPP protocol does not cover what happens when access rules change *after* subscriptions are already in progress. Under the current CPP scheme, no provision for sending a similar "failure to subscribe" is supported for subscriptions are currently in progress. Even if a watcher may succeed in subscribing for a specified duration, a presentity may decide later, before the subscription expires, that he needs to keep his presence information private. Clearly, privacy necessitates that no further notifications be sent to the subscriber; however, if no message is sent at all to the subscriber, then he is left

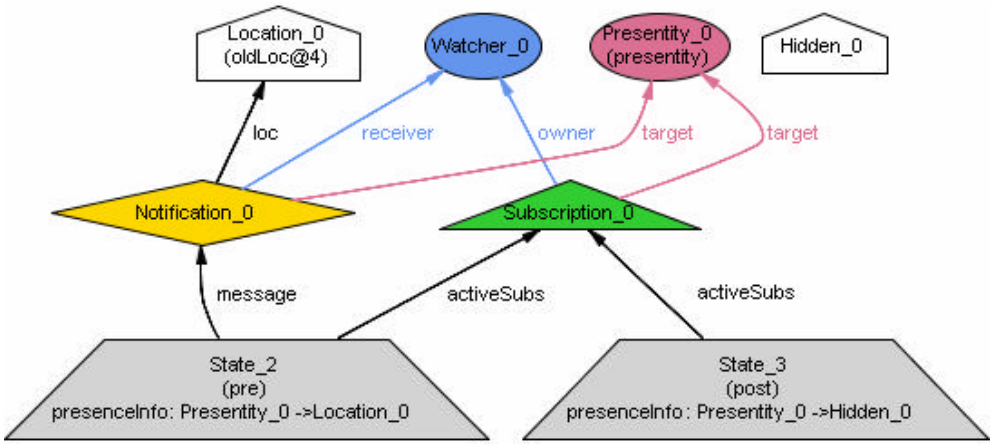

Figure 3: The CPP protocol leaves unspecified what happens if `Watcher_0` is subscribed to `Presentity_0`, but `Presentity_0` manipulates his access permissions and decides to hide. Is the watcher notified that the presentity is hidden, which may step upon privacy issues, or is the watcher left in the dark with the potentially out-of-date `PresenceNotification_0`?

with his most recent notification, which can potentially be inaccurate and out-of-date. Figure 3 illustrates this dilemma.

The only existing options for communicating the change in access rules are by invoking response operations or by sending notifications, both of which engender some problems and inconsistencies. As described on page 4 of the CPP protocol, response operations contain a nonce called the TransID attribute which corresponds to the TransID of the subscription to which it is responding. Utilizing response operations to notify changes in access rules thus necessitates the presence server storing all the TransID's of all active subscriptions, data which otherwise need not be maintained. Moreover, describing the communication of access rules as responses would be inaccurate since the subscribers have made not actually made a request for that information.

On the other hand, the alternative to communicate the change in access rules via sending notifications would be inconsistent with RFC2778's definition of a Notification operation on page 10:

> "NOTIFICATION: a message sent from the PRESENCE SERVICE to a SUBSCRIBER when there is a change in the PRESENCE INFORMATION of some PRESENTITY of interest, as recorded in one or more SUBSCRIPTIONS."

The inconsistency arises due to the fact that the presence information itself has not changed; only the *access* to that information has changed. Moreover, even if the CPP protocol chooses to utilize the notification operation for the purpose of informing subscribers of changed access rules, it is unclear how the change can be expressed using the notification attributes available in CPP: watcher, target, TransID, and presence information.

Hence, the following questions are broached:

1. What happens if a target decides to hide his presence information or otherwise changes his access control while subscriptions are in progress?
2. Will changing access rules generate a notification as in the case of changing locations?
2. Should Watchers be notified that a presentity is hidden? Or should the fact that the presentity is hidden be private and itself be hidden from the Watcher?

4.5 Asynchronous Notifications

The CPP protocol implicitly assumes that all server/client communication operates instantaneously and does not address issues regarding server overload or limited bandwidth, both of which may cause messages to be delayed. Clearly, this simplifying assumption will not always hold, and the order and accuracy of certain time-dependent operations may become compromised. Provisions must be made to address the following issues:

1. If a presentity undergoes more than one state change, either by changing location or by manipulating access rules, before the presence service is able to send the information to any subscribers, does the subscriber then receive a notification regarding the most recent status change or does he also receive backlogged notifications, which may have become out of date?
2. If a watcher sends more than one request before the presence service is able to respond, does the watcher subsequently receive responses for just the most recent message or for all the messages?
3. Does the presence service need to support the maintenance of a queue of messages to be sent? If not, how will the order of backlogged messages be guaranteed?

5 Conclusion

This case study of the Common Profile for Presence protocol illustrated the benefits of software modeling in design. By using the Alloy Analyzer as a design aid, I successfully identified inconsistencies between two apparently similar design documents and illuminated a number of issues that either escaped the CPP designers or that possessed sufficient ambiguity to warrant further refinement. Many of the design issues and inconsistencies of the protocol documented in this paper would not have been revealed this early in the design process without the lightweight modeling made possible by the Alloy Analyzer. I hope that the Instant Messaging and Presence Protocol working group will find this analysis useful in refining the CPP protocol.

6 Acknowledgements

I would like to thank Professor Daniel Jackson for discussing the case study with me and providing feedback on both my Alloy model and on this written report.

7 Appendix

Attached below is the Alloy model for impp/cpp.

```
/**
 * A model of the IMPP Common Profile for Presence
 *
 * This model is based on the following two drafts:
 *  Common Profile for Presence
 *  Model for Presence and Instant Messaging
 *
 * They can be found at http://www.ietf.org/html.charters/impp-charter.html.
 *
 * In a typical interaction, a Watcher application attempts to
 * subscribe to a Presence Service to receive notifications
 * regarding the presence information, i.e. location, of a
 * Presentity. If the subscription is successful, the Presence
 * Service notifies the Watcher of the Presentity's location;
 * it continues to send notifications whenever the Presentity's
 * presence information changes.
 *
 * Differences between this alloy model and the paper model:
 *
 * -- TransIDs and SubscriptionIDs have been ignored in this model as
 *    low-level details.
 * -- The Response Operation has been left out because it communicated
 *    no presence information and carried only a status flag.
 * -- To represent the manipulation to access privileges to so that one's
 *    presence information is not released, a Presentity moves into a
 *    Hidden location.
 * -- Durations in Subscriptions have been replaced with an expiry relation
 *    that points to the expiration time.
 *
 * Simplifying assumptions made in the alloy model:
 *
 * -- The Presence Service is considered to be a centralized server that
 *    handles all the various requests. It is implicitly modeled through
 *    the sequence of transitions of the Server's States: all information
 *    regarding presence and location is assumed to be contained within
 *    those states.
 */

module impp/cpp

open std/ord[State]

sig Presentity {}

sig Watcher {}

sig Location {}
sig Hidden extends Location {}

sig Subscription {
  owner: Watcher,
  target: Presentity
}

sig Notification {
  receiver: Watcher,
  target: Presentity,
  loc: Location
}
```

```
-- State represents a snapshot of the Presence Service at some point in time
sig State {
  -- mapping of each presentity to current location
  presenceInfo: Presentity -> some Location,
  -- notification message to be sent, if presenceInfo just changed
  //message: sole Notification,
  activeSubs: set Subscription
}

fact BasicConstraints {
  all s: State |
    let n = s.message | n.loc = currentLoc(s, n.target)
  Notification in State.message
  Subscription in State.activeSubs
}

// event modeling
abstract sig Event { pre, post: State, presentity: Presentity }
sig MoveEvent extends Event { newLoc: Location }
sig HideEvent extends Event {}
sig SubscribeEvent extends Event { subscriber: Watcher}
sig UnsubscribeEvent extends Event { unsubscriber: Watcher}
sig NotifyEvent extends Event { recipient: Watcher, changedLoc: Location }

/************************************************
 ** OPERATIONS
 ************************************************/

pred move(s, s': State, p: Presentity, l: Location) {
  let pres = s.presenceInfo, pres' = s'.presenceInfo {
    l not in Hidden
    p->l not in pres
    let oldLoc = p.pres | pres' = pres - p->oldLoc + p->l
  }
}

pred hide(s, s': State, p: Presentity) {
  let pres = s.presenceInfo, pres' = s'.presenceInfo {
    some Hidden
    p in pres.(Location - Hidden)
    some oldLoc: p.pres | pres' = pres - p->oldLoc + p->Hidden
  }
}

pred subscribe(s, s': State, w: Watcher, p: Presentity) {
  some sub: Subscription + s.activeSubs {
    sub.owner = w
    sub.target = p
    s'.activeSubs = s.activeSubs + sub
  }
}

pred unsubscribe(s, s': State, w: Watcher, p: Presentity) {
  some sub: s.activeSubs {
    sub.owner = w
    sub.target = p
    s'.activeSubs = s.activeSubs - sub
  }
}

pred notify(s: State, w: Watcher, p: Presentity, l: Location) {
  some n: Notification {
    n = s.message
    n.receiver = w
    n.target = p
    n.loc = l
  }
}
```

```
/************************************************
 ** BASIC CONDITIONS
 ************************************************/

pred subscribed(s: State, w: Watcher, p: Presentity) {
  some sub: s.activeSubs | sub.owner = w && sub.target = p
}

pred hidden(s: State, p: Presentity) {
  some currentLoc(s, p)
  currentLoc(s, p) in Hidden
}

fun currentLoc(s: State, p: Presentity): set Location {
  p.(s.presenceInfo)
}

/************************************************
 ** CPP PROTOCOL
 ************************************************/

pred NotifySubscriptionChanges() {
  all s: State - last() {
    let s' = next(s) {
      all p: Presentity, w: Watcher |
        (subscribe(s, s', w, p) || unsubscribe(s, s', w, p)) && !hidden(s,p) =>
              some s'': nexts(s') | notify(s'', w, p, currentLoc(s, p))
    }
  }
}

pred NotifySubscribersOfPresenceChanges() {
  all s: State - last() {
    let s' = next(s) |
      all sub: s.activeSubs, l: Location - Hidden |
        move(s, s', sub.target, l) =>
          (some s'': nexts(s') | notify(s'', sub.owner, sub.target, currentLoc(s', sub.target)))
  }
}

// if message operation occurs,
// then either location change, subscription, or unsubscription occurred some time before
pred NoNotifyUnlessSubscriptionOrLocationChange() {
  all s'': State - first(){
    no s''.message ||
    let n = s''.message |
      some s': prevs(s'') | let s = prev(s') |
        move(s, s', n.target, n.loc) && subscribed(s, n.receiver, n.target) ||
        subscribe(s, s', n.receiver, n.target) ||
        unsubscribe(s, s', n.receiver, n.target)
  }
}

fact NotificationRules {
  NotifySubscriptionChanges()
  NotifySubscribersOfPresenceChanges()
  NoNotifyUnlessSubscriptionOrLocationChange()
}


/************************************************
 ** OPERATIONAL SEQUENCE: state -> event -> state
 ************************************************/
pred init(s: State) {
  no s.message &&
  no s.activeSubs
}
```

```
// event transitions: an event links a pair of states together
pred trans(s, s': State) {
  some e: Event {
    e.pre = s
    e.post = s'

    (e in MoveEvent && move(e.pre, e.post, e.presentity, e.newLoc) &&
      updateSubscriptions(e.pre, e.post)) ||
    (e in HideEvent && hide(e.pre, e.post, e.presentity) &&
      updateSubscriptions(e.pre, e.post)) ||
    (e in SubscribeEvent && subscribe(e.pre, e.post, e.subscriber, e.presentity) &&
      samePresenceInfo(e.pre, e.post)) ||
    (e in UnsubscribeEvent && unsubscribe(e.pre, e.post, e.unsubscriber, e.presentity) &&
      samePresenceInfo(e.pre, e.post)) ||
    (e in NotifyEvent && notify(e.post, e.recipient, e.presentity, e.changedLoc) &&
      samePresenceInfo(e.pre, e.post) && updateSubscriptions(e.pre, e.post))
  }
}

pred link(s, s': State, e: Event) {
  e.pre = s
  e.post = s'
}

pred updateSubscriptions(s, s': State) {
  s'.activeSubs = s.activeSubs
}

pred samePresenceInfo(s, s': State) {
  s.presenceInfo = s'.presenceInfo
}


// force exactly one thing to happen b/w each state
fact OperationSeq {
  init(first())
  all s: State - last() { let s' = next(s) | trans(s, s') }
  all e: Event | e.post = next(e.pre)
  no disj e, e': Event | e.pre = e'.pre && e.post = e'.post

  // match up Notifications with NotifyEvents and Notify Operations
  all n: Notification | some e in NotifyEvent |
    notify(e.post, e.recipient, e.presentity, e.changedLoc)
  all n: Notification | one s: State | notify(s, n.receiver, n.target, n.loc)
}

/************************************************
 ** ANALYSES
 ************************************************/

pred BasicSubscription() {
  some p: Presentity, w: Watcher {
    some s: State | subscribe(s, next(s), w, p)
  }
}

pred BasicNotification() {
  some p: Presentity, w: Watcher {
    some s: State | subscribe(s, next(s), w, p)
    some s: State, l: Location | move(s, next(s), p, l) && subscribed(s, w, p)
  }
}

pred HidingLeadsToNotification() {
  some p: Presentity, w: Watcher, s: State |
    subscribed(s, w, p) && hide(s, next(s), p)
}
```

```
pred DoubleMove() {
  some p: Presentity, s: State, w: Watcher |
    let s' = next(s) | let s'' = next(s') {
      subscribed(s, w, p)
      some l: Location | move(s, s', p, l)
      some l': Location | move(s', s'', p, l')
    }
}

assert UpToDateNotification {
  let lastTwo = last() + prev(last()) |
    all s: State - lastTwo, p: Presentity, w: Watcher |
      let s' = next(s) |
        subscribe(s, s', w, p) && no Hidden =>
          some s'': nexts(s') | notify(s'', w, p, currentLoc(s'', p))
}

pred MultiLocatedPresentity() {
  some disj loc, loc': Location - Hidden, s: State, p: Presentity {
    p->loc in s.presenceInfo && p->loc' in s.presenceInfo
    let s' = next(s) |
      some w: Watcher | subscribe(s, s', w, p)
  }
}

pred UnsubscribeLeadsToNotification() {
  some s: State - last()|
    let s' = next(s) |
      some p: Presentity, w: Watcher |
        unsubscribe (s, s', w, p)
  no Hidden
}

pred Example () {
  let first = first() {
    let second = next(first) {
      let third = next(second) {
        let fourth = next(third) {
          let fifth = next(fourth) {
            some e: SubscribeEvent | link(first, second, e)
            some e: MoveEvent | link(second, third, e)
            //some e: NotifyEvent | link(second, third, e)
          }
        }
      }
    }
  }
  no Hidden
}

run BasicSubscription for 1 but 2 Event, 3 State
run BasicNotification for 1 but 4 Event, 5 State
run HidingLeadsToNotification for 1 but 2 Location, 3 Event, 4 State
run HidingLeadsToNotification for 1 but 2 Location, 4 Event, 5 State
check UpToDateNotification for 4 but 5 State

run MultiLocatedPresentity for 5 but 4 Event, 1 Presentity, 1 Watcher
run DoubleMove for 5 but 4 Event, 1 Presentity, 1 Watcher
run DoubleMove for 6 but 5 Event, 1 Presentity, 1 Watcher
run UnsubscribeLeadsToNotification for 3
run UnsubscribeLeadsToNotification for 4 but 2 Location, 1 Watcher, 1 Presentity
run UnsubscribeLeadsToNotification for 5 but 2 Location, 1 Watcher, 1 Presentity

run Example for 1 but 2 State
run Example for 2 but 3 State
run Example for 3 but 4 State
run Example for 1 but 5 Event, 1 SubscribeEvent, 1 MoveEvent, 2 NotifyEvent, 6 State
```