Modeling Call Forwarding in Alloy

Michel Lambert
Massachusetts Institute of Technology
450 Memorial Drive, J217
Cambridge MA, 02139
mlambert@mit.edu

Abstract:

Call forwarding in telephone systems allows for a variety of configurations in redirecting telephone calls. This paper describes the results of using Alloy, a structure-modeling tool, to visualize and find constraints in some models of the call-forwarding system. First, an example involving delegate forwarding and follow-me forwarding is described and analyzed. Next, an orthogonal model involving the time-changing nature of people availability is used to analyze other conditions in the telephone system. Finally, the results are analyzed to see if they have an impact on the real world.

Background and Motivation:

While modern call forwarding systems are quite powerful these days, the majority of them still retain the simple constructs from which they were derived. A basic call forwarding system includes the two notions of delegate forwarding and follow-me forwarding. Delegate forwarding is used to assign your incoming phone calls to another person, who will be responsible for handling your calls. Follow-me forwarding is used when calls you will still be answering your incoming phone calls, albeit in a different location. The delegate relation is transitive, whereas the follow-me relation is not.

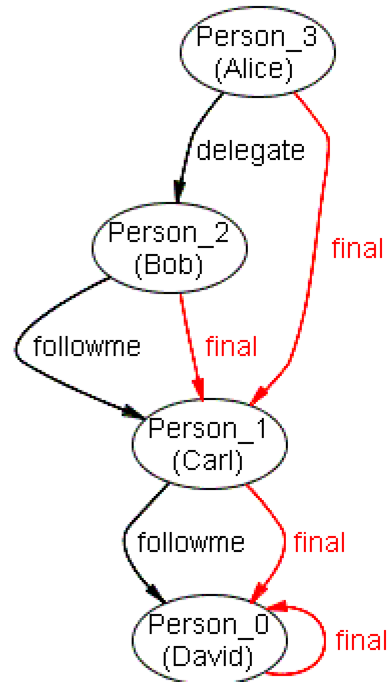

Figure 1

Take the hypothetical case of Alice, Bob, Carl, and David. Alice is going on vacation, and so she closes off her portion of the building, delegating her calls to Bob. Carl is working across the company in David's office working to finalize the specs for the company's latest project, and sets up a follow-me forward to David's office. Bob, unable to get into his office due to a misunderstanding, and is working from Carl's vacated office, and so sets up a follow-me forward to his temporary office. In this simple

example, all of Alice and Bob's calls will go to Carl's office, whereas Carl and David's calls will end up in David's office. A diagram of this setup can be found in Figure 1.

In addition, more complex call forwarding systems allow for more complex forwarding situations, such as delegate-always, delegate-busy, and delegate-no-answer. These conditional delegations, along with their follow-me counterparts, increase the chance of complex interactions creating undesirable configurations of a system. Failure of a call-forwarding system results can easily result in undesirable situations. In many cases, it is possible to have situations where an incoming call gets stuck in a forwarding cycle, or fails to get answered at all. This research project intended to analyze call forwarding systems to discover kind of constraints are needed to ensure that the configuration stays in a valid state.

Summary and Evaluation:

Using Alloy, telephone call forwarding configurations were analyzed for weaknesses and used to find solutions to prevent inconsistencies. In analyzing these systems, two different alloy models were used.

Primary Model:

The first model allowed configurations such as the simple Alice, Bob, Carl, and David example given above. It had the concept of a Person, and a State that contained the delegate and follow-me relations at that point in time. Calls did not physically exist, but rather found their destination in an instantaneous amount of time through Alloy functions that gave the destination Person. By analyzing this model, a few inconsistent configurations were found.

First, the constraint that all configurations had a single destination for every call did not hold up. If a delegate cycle were created in the system, each phone call would cycle forever between the people, unable to find a destination telephone to handle the call. There were two ways that I found to solve this problem. The simplest was to simply guarantee that a person did not appear in his or her own delegation chain (*all s: State, p: Person | p not in ^(s.delegate)[p]*). While this is a valid solution to the problem, it might not adequately explain to the user why their particular request to create a delegate forwarding did not work. A second solution to this problem is to declare an ordering over Persons, as often exists in a company. With a total ordering, one can create the constraint that the only valid delegations are to people farther down in the order. The total ordering explicitly ensures that no such cycles can be generated. Real companies usually have a partial ordering with employees. If delegations to unordered employees are disallowed, then a partial order also satisfies this problem because it has degenerated into a total order among a subset of the employees of the company.

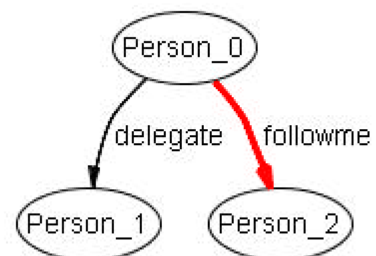

Figure 2

In addition to the above constraint, it was found that the telephone call-forwarding configurations created by Alloy sometimes contained unnecessary information. For example, in Figure 2, the call-forwarding algorithm described above never traverses the thicker follow-me link. This small problem with telephone configuration systems is easily fixed by ensuring that every Person can either have a follow-me link or a delegate link, but not both.

The first model also allowed for states that changed their configuration tables with each transition. However, allowing states to change with a transition did no produce any additional violations of the constraints. It was merely stringing multiple states together with no coherency.

Secondary Model:

The knowledge that there were certain inconsistencies that would only show up with the progression of a telephone call through the forwarding system necessitated the need of a second model. This model only had the concept of a delegate relation to make things simpler, but made up for it with the concept of a Call signature, which progressed through the call-forwarding graph over state transitions. It may not seem realistic to allow non-finite amount of time for a call to traverse, when that same amount of time can allow multiple people to change their delegation relations. However, some calls may take a long time to reach a final destination, if they must wait through delegate-no-answer or delegate-busy conditions. As such, it is believed that the model can still accurately reflect possible situations.

With this model, it became much easier to see the path a call traveled through in the call forwarding graph during its lifetime. It was found that even if no loops were allowed at any given state, that calls could still traverse indefinitely as the forwarding graph changed while the call was moving. If we took the second approach mentioned above, such that delegation relation does not point to any person higher in the total order, then it also eliminated the infinitely traversing graphs.

Other situations that were found to occur included a call that terminated its search while failing to find a telephone. In real life situations, this usually cannot occur, since every person, by default, has a delegate-no-answer and/or delegate-busy to their respective answering machine. Since answering machines are never busy or unavailable, this solves this particular problem. (Of course, if you have a limited number of lines on your answering machine, then you must worry about this particular situation as well.)

In conclusion, when one is setting up a telephone call forwarding system, one must be careful to ensure that all cases are allowed for, and that there are backups in place such that no incoming calls get stuck in the internal telephone system void, or get dropped without a proper resolution. With proper alloy models, and some additional constraints that reflect the setup of the company, this is possible.

Lessons Learned:

When starting to work on this particular modeling problem, I thought it would be too simple and trivial to find any useful information. But after playing with a few different ways of modelling the system, I found it to be complex enough to warrant the project, and did proceed to find interesting scenarios.

Alloy, as a tool, was quite helpful. I was disappointed in it's lack of supporting recursive functions, although that is understandable given it's approach of inlining functions. This made me realize that Alloy was quite successful at modelling the progress and transitions in state of a given algorithm, but was not particularly adept at allowing one to implement the algorithm.

In the example of the call forwarding models, the second model included the transitions of a call as it travelled through the system. The first model, was intended to allow the result of the algorithm to be generated via functions, which returned the result of the call given the initial Person that was called. While I was eventually able to succeed in this, (by returning all non-delegating nodes in the transitive closure of the delegating nodes,) t did require a rethinking of the problem in set theory before I was successful.

There were a few situations that became slightly annoying when using Alloy, including the NotEnoughMemoryException. Many times I would try to evaluate some constraints, only to receive the error. If I immediately re-ran the command, then it would proceed to check the constraints just fine.

Finally, I found the help of diagnostic relations helpful. In the case of the primary model below, I created three additional relations that were merely byproducts of the other relations. Named final, final_del, and final_fol, they were helpful in modelling the system's half-completed results, to verify that my functions were correct in their data.

The problem with these diagnostic relations came when attempting to edit instances to debug my constraints. In order to avoid filling out these particular constraints which were byproducts of my primary relations, I had to comment out the code and recompile. I would have been nice to have ternary checkboxes in the Edit Instance screen which would allow one to specify On, Off, and Figure-it-out. This would allow you to partially specify an instance, and have Alloy figure out which constraints were violated by your partial completion of the instance, or say that nothing was violated by those alone.

Primary Model

```
module samples/telesystem
open std/ord

sig Person {}

sig State {
	delegate: Person->?Person,
	followme: Person->?Person,
	// these three are variables used to visualize
	// the results of the translation functions in the Graph
	final_del: Person->Person,
	final_fol: Person->Person,
	final: Person->Person
	}

fact SetupInfo {
	all s: State, p: Person {
		s.final_del[p] = DelegateDest( s, p )
		s.final_fol[p] = FollowmeDest( s, p )
		s.final    [p] = Dest        ( s, p )
		}
	}

fact SomePeople {
	Person = univ[Person]
	State = univ[State]
	}

fun DelegateDest( s: State, p: Person ): Person {
	result = *(s.delegate)[p] - (s.delegate).( *(s.delegate)[p] )
	}

fun FollowmeDest( s: State, p: Person ): Person {
	result = if some s.followme[p] then s.followme[p] else p
	}

fun Dest( s: State, p: Person ): Person {
	result = FollowmeDest( s, DelegateDest( s, p ) )
	}

//New Constraints Needed
//fixes NotAllDelegate
fun NoCircularDelegateReference() {
	all s: State, p: Person | p not in ^(s.delegate)[p]
	}
//fixes AllLinksUsed
fun NoDelegateAndFollowme() {
	all s: State, p: Person {
		not (some s.delegate[p] and some s.followme[p])
		}
	}


//Checks and Assertions
fun OneDestination() {
	all s: State, p: Person | one Dest( s, p )
	}

assert OneDestination_NoLoops {
	NoCircularDelegateReference() => OneDestination()
	}
check OneDestination_NoLoops for 3 but 1 State

// this solves the OneDestination problem as well
assert OneDestination_TotalOrdering {
	( all p: Person | State.delegate[p] in OrdPrevs(p)-p )
	=> OneDestination()
	}
```

```
check OneDestination_TotalOrdering for 3 but 1 State

assert NotAllDelegate {
        NoCircularDelegateReference() =>
        ( some s: State, p: Person | p = DelegateDest( s, p ) )
        }
check NotAllDelegate for 3 but 1 State


fun GetUsedFollowmeMappings( s: State ): Person->Person {
        all p: DelegateDest( s, Person ) {
                p->FollowmeDest( s, p ) in result
                }
        }

assert AllLinksUsed {
        ( NoCircularDelegateReference() && NoDelegateAndFollowme() ) =>
        =>
        ( all s: State {
                all fm: s.followme {
                        fm in GetUsedFollowmeMappings( s )
                        }
                } )
        }
check AllLinksUsed for 3 but 1 State

fun BaseChecks() {
        NoDelegateAndFollowme()
        NoCircularDelegateReference()
        }
//Running Examples

fun AllFollow() {
        BaseChecks()
        all s: State, p: Person | p != FollowmeDest( s, p )
        }
run AllFollow for 4 but 1 State

fun AllDelegateButOne() {
        BaseChecks()
        all s: State {
                one p: Person | p = DelegateDest( s, p )
                }
        }
run AllDelegateButOne for 3 but 1 State


fun ChangePerTransition() {
        BaseChecks()
        all s: State - Ord[State].last {
                one p: Person {
                        OrdNext(s).delegate[p] != s.delegate[p]
                        or
                        OrdNext(s).followme[p] != s.followme[p]
                        }
                }
        }
run ChangePerTransition for 3 but 2 State

/* Disabled so all people don't show up with these names
part sig Alice, Bob, Carl, David extends Person {}
fun ABCDExample() {
        BaseChecks()
        #Alice = 1
        #Bob = 1
        #Carl = 1
        #David = 1
        State.followme[Bob] = Carl
        State.followme[Carl] = David
        State.delegate[Alice] = Bob
        }
```

```
run ABCDExample for 4 but 1 State
*/
```

Secondary Model

```
module samples/telesystemtime
open std/ord


sig Person {}
sig AnsweringMachine extends Person {}

sig Status {}
part sig Available, Unavailable extends Status {}
fact StatusFact {
        #Available = 1
        #Unavailable = 1
        #Status = 2
        }

sig Call {}

sig State {
        entitystatus: Person->!Status,
        entityforward: Person->!Person,
        callstatus: Call?->?Person,
        callscompleted: Call->?Person,
        callsfailed: Call->?Person
        }

fact CallLogistics {
        all c: Call | some State.callstatus[c]
        no Ord[State].first.callscompleted[Call]
        no Ord[State].first.callsfailed[Call]
        }

fact AnsweringMachineConstraints {
        State.entitystatus[AnsweringMachine] in Available
        no State.entityforward[AnsweringMachine]
        }

fact useAll {
        univ[Person] = Person
        univ[Call] = Call
        univ[Status] = Status
        }

fact AllPeopleHaveStatus {
        all p: Person, s: State | one s.entitystatus[p]
        }

fact CallOnlyInOnePlace {
        all s: State, c: Call {
                #s.callstatus[c] + #s.callscompleted[c] + #s.callsfailed[c] < 2
                }
        }


fun Transition( s, s': State ) {
        //Progress all calls, or end them
        all c: (s.callstatus).Person {
                s.entitystatus[ s.callstatus[c] ] = Unavailable => {
                        some s.entityforward[ s.callstatus[c] ] =>
                                s'.callstatus[c] = s.entityforward[ s.callstatus[c] ]
                        no s.entityforward[ s.callstatus[c] ] => {
                                no s'.callstatus[c]
                                s'.callsfailed[c] = s.entityforward[ s.callstatus[c] ]
                                }
                        }
                s.entitystatus[ s.callstatus[c] ] = Available => {
```

```
                        no s'.callstatus[c]
                        s'.callscompleted[c] = s.entityforward[ s.callstatus[c] ]
                        s'.entitystatus[ s.callstatus[c] ] = Unavailable
                        }
                }
        s.callsfailed[Call] in s'.callsfailed[Call]
        s.callscompleted[Call] in s'.callscompleted[Call]
        }

fact Constraints {
        all s: State - Ord[State].last {
                Transition( s, OrdNext(s) )
                }
        }

// No complete failure loops in one state
fun NoLoopsInState() {
        all s: State | all p: Person {
                p not in ^(s.entityforward)[p] or
                some p': ^(s.entityforward)[p] | s.entitystatus[p'] = Available
                }
        }

// At most one new call
fun OneNewCallPerTick( s, s': State ) {
        //Add at most one new call each transition
        one p: Person | s'.callstatus[Call] - p in s.callstatus[Call]
        }

// One person must change their status each time
fun OneStatusChangePerTick( s, s': State ) {
        one p: Person | s'.entitystatus[p] != s.entitystatus[p]
        }

// Only one person can change their status each time
fun MaybeOneStatusChangePerTick( s, s': State ) {
        sole p: Person | s'.entitystatus[p] != s.entitystatus[p]
        }

// Do not allow changes to the forwarding table
fun NoForwardingChangesPerTick( s, s': State ) {
        all p: Person | s'.entityforward[p] = s.entityforward[p]
        }


fun LoopWithoutProblems_FullLengthCallTraversal() {
        NoLoopsInState()
        (Ord[State].first.callstatus).Person = Call
        some OrdPrev( Ord[State].last ).callstatus[Call]
        (Ord[State].last.callscompleted).Person = Call
        all s: State {
                some p: Person | ^(s.entityforward)[p] = Person
                }
        all s: State - Ord[State].last {
                NoForwardingChangesPerTick( s, OrdNext(s) )
                MaybeOneStatusChangePerTick( s, OrdNext(s) )
                }
        }

run LoopWithoutProblems_FullLengthCallTraversal for 3 but 3 State, 1 Call, 2 Status

//Demonstrates that with 1 person, one can have infinite loops if direction is allowed to
oneself
fun RunLongTime() {
        some Ord[State].first.callstatus[Call]
        some Ord[State].last.callstatus[Call]
        all s: State - Ord[State].last {
                NoForwardingChangesPerTick( s, OrdNext(s) )
                MaybeOneStatusChangePerTick( s, OrdNext(s) )
                }
        }
```

```
run RunLongTime for 1 but 6 State, 1 Call, 2 Status

//Demonstrates that with 1 person, one doesn't need infinite loops if direction is
allowed to oneself
fun RunShortTime() {
        some Ord[State].first.callstatus[Call]
        some Ord[State].last.callscompleted[Call]
        all s: State - Ord[State].last {
                NoForwardingChangesPerTick( s, OrdNext(s) )
                MaybeOneStatusChangePerTick( s, OrdNext(s) )
                }
        }

run RunShortTime for 1 but 6 State, 1 Call, 2 Status

//Demonstrates that with 4 person, and no loops at any point in time, one can still have
infinite loops for the Call
fun RunLongTimeWithNoLoops() {
        NoLoopsInState()
        some Ord[State].first.callstatus[Call]
        some Ord[State].last.callstatus[Call]
        all s: State - Ord[State].last {
                NoForwardingChangesPerTick( s, OrdNext(s) )
                }
        }

run RunLongTimeWithNoLoops for 2 but 4 State, 1 Call, 2 Status

assert RunLongTimeWithNoLoops_AndOneStatusChangePerTick {
        RunLongTimeWithNoLoops()
        all s: State - Ord[State].last {
                MaybeOneStatusChangePerTick( s, OrdNext(s) )
                }
        some s: State - Ord[State].last | Ord[State].last = s
        }

check RunLongTimeWithNoLoops_AndOneStatusChangePerTick for 3 but 12 State, 1 Call, 2
Status
```