Agent-Based Modeling of Covert Social Groups, Applying Software Engineering Patterns and Planning for Cluster Computation

Carl A. B. Pearson[1,*], Edo Airoldi[2], Edward Kao[2], Burton Singer[1],
1 Emerging Pathogens Institute, University of Florida, Gainesville, FL, USA
2 Statistics, Harvard University, Cambridge, MA, USA
∗ **E-mail: cap10 {at} ufl {dot} edu**

Abstract

We discuss covert social network analysis (SNA) in the larger scientific context, highlight pitfalls, and propose what we view as necessary analysis that seems absent. Because SNA typically occurs on aggregated observations (e.g., ties are across types and times), and furthermore covert SNA involves at least some (and often severe) censoring of those observations, it is critical that the implications of that aggregation and observation process be considered.

We demonstrate an implementation framework for performing that kind of analysis, based on a distributed, agent-based model. We also apply past software engineering insights to show how they can make for more natural (and therefore comprehensible and compartmentally testable) agent-based models in this framework.

We close with discussion of how this framework might be used in a practical application, and how it is might be pieced together with other analytical tools.

Covert social groups are difficult to observe directly, by definition: members either actively thwart discovery or leverage passive features of their surrounding population to avoid detection. They watch the watchers as well, and increasing effort to reveal them should be expected to encourage adaptation. Yet, as impossible as it is for outsiders to perfectly see the covert group, it is equally impossible for these groups to perfectly hide and accomplish anything of note.

Introduction

For investigators ranging from anthropologists to law enforcement, the desire to find, study, and modify the behavior of groups operating at the edge of the observable is a principle concern. In the meta-theatre of the ironic, various media outlets have responded to public interest by applying this lens to the activities of intelligence organizations world-wide – themselves desperate to undertake this very task against other clandestine actors. The revealation of these activities has sparked the symmetric concern: being able to operate clandestinely in an age of ubiquitous monitoring. Criminal organizations have long appreciated the value of operating in secret, as have groups subject to State-sponsored abuse or industrial espionage, and time will tell whether or not this spark catches fire in general behavior.

The underlying drive for these opposed efforts is the implacably expanding byte trail. Once, the recorded information of an entire life might have amounted only to a few bytes – e.g., parish records on births and deaths, perhaps including morbidity. Now information era tools produce a near endless supply of 1s and 0s. In some cases, this production is a side effect of organized commercial instinct – Google monetizing search through targeted advertisement, for example. In other cases, individuals are voluntarily advancing their own brand – Instagram "selfies", e.g. However, in many cases this bitwise production is an engineering consequence of the associated technology. E.g., cellular phones transmit constant location data, financial systems are increasing just digital ledgers substituting the certainty of physical exchange with sophisticated book-keeping to ensure accounts are square, and of course any use of the internet produces veritable bit contrails. This rate of production far exceeds the personal processing capability of any practically-sized team of analysts.

Hence, these teams employ computer-based, heuristic filtration to decide which data to record, to review, and to obtain. We avoid saying "algorithmic" at this point: "algorithmic" implies strong, logically inevitable conclusions from inputs. Like "algorithmic" economic bubble prediction, we side more with Fama than Shiller.

Many such filters are based on social network descriptions. Social networks are a compact way to represent measured relations in a population. That compression, however, often involves irreversible aggregations: in strength – e.g., co-employment as measured by working at the same university versus the same department versus in same the laboratory; in type – e.g., engaging in commerce versus attending the same church versus having married relations; and in time – e.g., co-appearance at a coffeeshop months ago versus attending classes together on a weekly schedule versus living together.

The mathematics may provide a formally guaranteed result from some particular input, but the more aggregation that occurs translating observations – themselves imperfect snapshots of actual phenomena – to representation, the more convincing evidence that must be provided before we should accept that those representations (and subsequent results from manipulating them) have meaning.

Given the real uncertainty, what these filters call for is testing and validation, but those present their own difficulties. Calling field testing "problematic" seems like a gross understatement; reference "truth" ranges from incomplete to deceptive, and experiments could have dangerous side effects. Even making use of intensely studied historical events is problematic: these offer no way to consider evolutionary behavior and technological innovation (without an additional layer of abstraction that would create an even more complex empirical challenge), even assuming the historical data are more than victor's embellishment.

Generating synthetic data seems like an obvious alternative. It allows for comparison across both detection and masking strategies, consideration of multiple background contexts, forecasting of risks and tradeoffs in a way that allows uncertainties, and in general providing a framework for imaginative assessment. Like all such flexible tools producing quantitative results, it has the subtle

downside of the simulator's biases being validated by numerical gospel; if one believes a particular strategy is effective – perhaps even with reasonable agreement for a particular time and situation – there would be a natural tendency to "adjust" scenario parameters until they indicated the success of strategy. Simple models are equally problematic, as they replace the parameter fudge-factor with the complementary assumptions fudge-factors. Thus, we must always be skeptical - of hordes of purely fitted parameters that do not have an attached mechanism we can validate the resulting numbers against, and of the assumptions buried simple models that may in fact be driving model outcomes.

A messy business all around, and we have not even gotten to the technically hard details.

Skeptical Simulation

If, as we believe, generating synthetic data for the purpose of testing covert SNA methods is the only pragmatic option, how should one proceed in light of all the obstacles?

As the point is to assess the conclusions of those methods against what is "actually" happening (*in silico*), on one end the process must obviously produce output that can be ultimately represented as a social network. Since any the subject SNA method should include the process of converting real observations to the represention, the output should be in the form of those observations. That part is conceptually straightforward, if not necessarily technically so.

The other end – what is "actually" happening, at least in simulation – is less conceptually straightforward, the problem being that we do not know all of what is actually happening. We propose here that researchers adopt a modeling philosophy that makes them open to criticism. That is, the model meets the following criteria:

- it is clear which parts are explicit (subject to fitting – e.g., rate constants) versus implicit (assumptions – e.g., interaction forms)
- it is clear what those explicit and implicit processes are doing
- it is clear where parts interact
- it is clear how the observation process interacts with the modeled system

Consider the covert SNA described by Bright, et al. of the social networks created by judge's sentencing comments indicating personal association in methamphetamine production enterprises. They provide the key components for the proposed modeling philosophy:

- a description of the subject process - e.g., acquisition of precursor, production in clandestine laboratory, distribution, as well as necessity of specialists
- a description of how that process is observed - e.g., law enforcement investigation and subsequent prosecution
- how the observations are flattened into a network representation - translation of judge's sentencing comments
- the specific SNA methods applied and desired inferences - e.g., various centralities and applications to intervention

With the exception of the actual SNA, however, these are "talking" models. We assert that to genuinely assess if the SNA based on judicial sentencing notes, we must explicitly understand how gang activity ultimately percolates into that ink. That it is necessary to turn the descriptive models into computational ones, while preserving (and hopefully enhancing) our ability to argue with each piece of that story and all the between-the-lines implications.

Features of Good Computational Model

What makes a good computational model? Modellers in the traditional hard sciences (physics, chemistry, biology, and assorted sub disciplines) have historically emphasized purely computational concerns, which is quite sensible for problems that have simpler conceptual models (i.e., those that are mathematically formalizable) but relatively intense numerical tasks. On the other hand, many non-academic software products deal with a quite sophisticated conceptual model with relatively simple numerical tasks, and that demand tends to push their development more towards programming and engineering concerns. We assert this domain – software engineering – is the better reference for social science modeling, and that the lessons of that discipline are more keenly relevant (though they are certainly relevant to the natural sciences, especially as the biological sciences deal with more of the heterogeniety intrinsics to that domain).

We will emphasize a few key lessons, some "ancient" wisdom (two plus decades old) and some a bit more recent. Good modeling is quite like good (scientific) story telling, with the exception that the story must be told by the code (though we include in "code" more than just what runs a particular simulation).

The overall goal is to make implementation choices that make the model itself communicate with other researchers. This means the right level of dividing up the model. Too little, and the rest of us have to comprehend the whole thing to comprehend a single atom of the story. Too much, and instead we have to go to quarks to understand an atom of the story.

This begins with choosing a setup (language, libraries, frameworks, etc.) that make this possible.

- literate programming
- design patterns - particularly emphasis on composition, decoration, adaptation over inheritance + achieving that with chain of responsibility

They provide a descriptive model of drugs production that appears nearly identical to most industrial enterprises: acquisition of materials, production, retailing, wholesaling, and regulatory "compliance" (bribing officials). Notably, the detail of this description and its quite reasonable analogy to regular enterprise.

There have been other attempts at such an approach [2].

In the following sections, we lay out the uses and abuses of such a framework. What makes for useful synthetic data sets? What are the appropriate measures for detection strategies on them? We motivate that discussion by inspiration from an over-simplified, network-based model of terrorism – a sub-group of the Salafi jihad networks as described by Sageman *et al.* [**?**] – and community organization and communication. Whether or not that work is an accurate description is not of particular concern. Their qualitative properties are enough of a defensible testbed for modeling the communication patterns of one covert group. We will point out where assumptions can be modified to identify different kinds of groups against a background population, since the behavior and structure of both the background and covert organizations are constantly evolving.

Overview of the Simulation Problem

Detecting a covert group is fundamentally about distinguishing the trace observable activity of that group from the activity of the background, bootstrapping that into guesses about the structure of the group, focusing the observation and distinguishing process based on that structure, and so on iteratively. The purpose of simulation is ultimately to measure performance in the observer-covert group competition.

Given that these aspects must be expressed in the model (with varying degrees of detail), and that this model is used by a team or communicated between researchers, the language for that expression must be both powerful (to capture complexity where necessary), but also comprehensible to communicate what in fact is the meaning of the universe in this model. Because these simulations are fundamentally about individuals and their behavior, we largely frame our discussion in terms of individuals, their internal state, and their interaction with others.

We can of course think in different terms. For example, we are considering a network model of the population and covert group embedded within it, with edges representing relationships between people. We might think of those relationships as being external state and correspondingly represent those relationships as external state in the simulation. E.g., we might represent the population as a graph rather than a bag of individuals, and in the simulation the edges belong to the graph object rather than individuals. As long as we recognize the translation between these perspectives, the choice boils down to what is most pragmatic relative to the science and simulation. If most of the questions answered by the simulation relate to graph measures – e.g., calculating various centrality measures – then the simulation-as-graph rather than simulation-as-individuals probably makes more sense.

Modeling the Covert Groups

Relative to the covert group, the simulation must model how the group members act and interact. Typically, that further entails modeling a few moving pieces: (i) the members internal state (including what other members they know and communicate with), (ii) how that internal state evolves relative to external forces or states, and (iii) what actions those members undertake based on their internal state (possibly in response to some specific external event).

Modeling the Background

Like the covert group, the background population is a (larger) group of individuals that act and interact. They have the same basic modeling requirements as the covert group, but they behave differently. This difference may be as simple as being distinct relative to observation process – e.g., their communications are more likely to be recorded – or it may be more noticeable like having fundamentally more diverse contacts, taking different kinds of actions, etc.

An alternative representation for the background would be to model it as a continuous entity instead of individuals in a network. This could be used to reduce the simulation size. However, this model would still need to create activity data that was consistent (in format) to that created by the covert group, because of the simulation requirements for observation.

Modeling Activity

Individuals create modeled activity based on their internal state. These actions correspond to (i) the observable events of specific interest (e.g., calls, financial transactions), (ii) generally observable events (e.g., travel, work, religious observations), and (iii) interactions that, while not externally observable, play a role in the evolution of internal state in other individuals either directly or by modifying the state of the world. These actions should occur at specific times in the simulation, at whatever resolution is appropriate to the model.

The covert group and background population should be different in their activity; either performing different activities or undertaking them at different rates or on different schedules.

The Observation Model

Activity is understood by the observing entities by an imperfect process. Depending on the situation modeled, that process might simply miss events. Alternatively, the error could be misunderstanding the type or content of the activity, or between what individuals and interaction is occurring. In general, the observing entities should be able to modify what is observed at what rates. Also, the observation process may itself alter the state of the world – i.e., the covert group may have the opportunity to respond to the observation process.

A Language Matching the Simulation Requirements: Scala

Scala [4] is an mixin-inheritance [1], object-oriented, functional language, which also includes an message-passing based model for parallel computing[1]. Indeed, Scala is used under the hood for the network simulation framework NetLogo.

Why do these traits make Scala particularly effective for the simulation requirements described above?

First, the covert group members and the background population share behavior (in the real and simulation sense), but are not necessarily simple extensions, which makes a mixin inheritance object system a natural fit for the model subjects.

Second, many interactions in an organization can be thought of as management directing labor to complete a task; in a functional paradigm, functions are themselves treated as objects. This translates natural to sending a function object from one simulated individual (in a leadership position) to another (in a subordinate position). Loosely adopting some of the other functional paradigms, as Scala does – emphasis on immutability, principally – provides practical advantages to concurrent programming, which is necessary for large scale simulations. Additionally, the functional emphasis bakes in other programming constructs to do with collection processing (e.g., iteration, transformation, filtering) that are less convenient in earlier languages.

Finally, the message-passing framework is an exact analogy to an individual / agent / actor - based model. Having this baked into the language makes it a feature more like control branches (e.g., `if`, `else`, `for`) or collections, than a more complicated syntax available only to advanced programmers (and often only via library support) rather than more phenomena- and model-focused scientists.

Practical Application

The practical application of these ideas is – for a simple simulation – as straightforward as stated.

Using the Scala actor framework, we simply define how each agent in the system responds to events, including the observing entities, and how they generate new events. We accomplish the first by appropriately defining `receive` and the handling of different `case`s of messages. We accomplish the second by having actors know about other actors, and having their internal state determine if they send events to these actors at any particular time.

In considering the salafi network (see fig. 1), we will observe a highly constrained set of activity: only communication, with all of that communication having been reduced to `Good` or `Bad`. The communication connections between individuals will be static, therefore we may set the social network at the outset of the simulation.

Figure 1. the salafi network

[1]The actor model is transitioning from the initial implementation to that developed by the Akka project [3]. We base our discussion on the Akka model. The most important difference is the notion of contexts and their switching and stacking: with a switching and stacking, an actor alters how it responds to events. This is a useful abstraction for representing evolving individual state.

In the code, we could define different kinds of actors – the covert leader, covert followers, background individuals – as different classes of simulation object. However, we instead take advantage of the context switching in the newer Scala actor framework. By approaching the simulation this way, we lay a good foundation for observing the dynamics of individuals – transitioning from background to covert member to covert leader – as well as the dynamics of relationship and group formation and dissolution.

One syntactical aside for the code the below sections: the Scala collections (like `Set`) define methods for adding and removing elements in terms of the natural mathematical operators (`+`, `-`) and for similar operations with groups of elements (`++`, `--`). Similarly, the names for message senders (`sender`), changing context (`context become ...`), et cetera are all quite clear. Given the informative, mixed-natural-and-mathematical-like syntax, we prefer actual code snippets to pseudocode to illustrate simulation steps. However, we have elided some declarations that enable the more informative syntax from the initial appearance in code.

The Covert Group

First, we define the covert group `Receive` context:

```
def plotter(
  // a plotter has superiors, collaborators, and subordinates
  // these groups are initially empty
  superiors:Plotters = empty,
  collaborators:Plotters = empty,
  subordinates:Plotters = empty
  /*, ... other args */) : Receive = {
    // ... to-be-defined behavior
    case _ => // ignore unmatched messages
}
```

We need to define how they adopt and evolve this context. Since we are not studying the organizational dynamics with this simulation, we can be phenomenologically loose for this part, though this approach means we can add monitoring that activity as part of future detection scenarios with minimal coding. The simulation runner will send a `Radicalize` message to the initial cabal, then a `Collaborate` message to the same group:

```
object Radicalize
case class Collaborate(group:Plotters)

def receive = {
  case Radicalize =>
    context become plotter
  case _ => // ignore other messages
}

def plotter(...) => {
  case Collaboration(_, group) =>
    context become plotter(collaborators = collaborators ++ (group - self))
  // ...
}
```

These leaders then need to recruit subordinates. In this particular simulation, we are using exactly the network from Sageman et al., so we will have the simulation runner inform each plotter whom they are recruiting with a `Subordinates` message. The leads will then send `Recruit` messages to those individuals and they will adopt the plotter context.

```
case class Subordinates(recruits:Plotters)
object Recruit

def plotter(...) = {
  case Subordinates(_, recruits) =>
    recruits foreach { recruit =>
      recruit ! Recruit
    }
  // ... other cases
}

def receive = {
  case Recruit =>
    context become plotter(superiors = Set(sender))
  // ... other cases
}
```

At this point, we have all the necessary messages to construct the given network. Those messages, based on the labels in Sageman et al. are covered in Appendix **??**. However, we still need to model how the plotters actually plot. We will assume that all plotting (i) originates with leadership, (ii) when intermediates receive direction, they work with their `collaborators` as well as delegating work, (iii) that directions occasionally result in feedback to higher-ups, and finally (iv) all of the "work" and direction is captured as passing `Bad` messages.

The simulation runs as a series of iterations, and during each time step a leader may initiate a new `Bad` message. Other members of the covert group circulate `Bad` messages subsequent to receiving a `Bad` message themselves, with different probabilities for sending messages to superiors, collaborators, and subordinates. Those probabilities do not depend on the origin of the received message (though this could be a useful mechanical tweak to the model).

Though those probabilities could be coded into the actors as part of their initialization, if they are set and adjusted by events then the actors (i) more accurrately reflect real behavior and (ii) the evolution of individuals occurs via a consistent simulation mechanism. The former provides a lever for expert insight and empirical study. The latter makes it easier to produce real-alike data – i.e., simulation data that has the form of real observations.

In our simplified example, we do not consider how "real" events would translate into adjusting a covert member's behavior. Instead, the simulation runner sends a message, giving us an entry point for later adjustment without frontloading too much model complexity. This represents the final step for how the plotters evolve in this simulation, so the following repeats the previous snippets with some condensing modifications to the code. The most notable one is making use of a `case class` to represent all of the parameters that form a plotter's state.

```
package object simactor { // defs to make implementation more natural
  type Plotters = Set[ActorRef]
  def empty : Plotters = Set.empty
  def group(list:ActorRef*) : Plotters = list.toSet
  type Probability = Double
}

object SimActor { // defines common elements for SimActors
  // MESSAGES ABOUT INTERNAL STATE
  object Radicalize                       // become a plotter
  object Recruit                          // work for a plotter
  case class Collaborate(group:Plotters)  // work with other plotters
  case class Subordinates(recruits:Plotters) // recruit subordinates

  object PType extends Enumeration { // plotter behavior probabilites
    type PType = Value
    val Initiate, // prob. of starting plot comms
        Report,  // prob. of reporting to superiors
          Collab,       // prob. of working w/ peers
          Delegate // prob. of delegating work
                 = Value
  }
  case class UpdateProb(changes:Map[PType.Value,Probability])

  // MESSAGES ABOUT EXTERNAL ACTIVITY
  object Bad
  object Good

  case class PlotterState( // the plotter state description
    superiors:Plotters = empty,
    collaborators:Plotters = empty,
    subordinates:Plotters = empty,
    probs:Map[PType.Value,Probability] = PType.values.map { (_, 0.0) } toMap
    // default each plotting probability 0.0
  )
}

class SimActor extends Actor {
  import SimActor._

  def receive = {
    case Radicalize =>
      context become plotter()
    case Recruit =>
      context become plotter(PlotterState(superiors = group(sender)))
    case _ => // ignore other messages
  }

  def plotter(ps:PlotterState = PlotterState()) : Receive = {
    import ps.{copy => change, _}
    def evolve(newPs:PlotterState) = context become plotter(newPs)

    {
       case Collaborate(group) =>
         evolve(change(collaborators = collaborators ++ (group - self)))
       case Subordinates(recruits) =>
         recruits foreach { recruit => recruit ! Recruit }
         evolve(change(subordinates = subordinates ++ recruits))
       case Recruit =>
         evolve(change(superiors = superiors + sender))
       case UpdateProb(changes) =>
         evolve(change(probs = probs ++ changes))
       case _ => // ignore other messages
    }
  }
}
```

Now we address how the plotters plot: if they receive a `Bad` message from within the covert group, they have some probability of continuing plot activity with their peers, some probability of delegating some work to subordinates (if they have any), and some probability of reporting to their supervisors (again if they have any). We will treat these activities as (i) independent, and (ii) the interactions with a particular group as a series of independent binomial trials. We are not aware of any mechanical explanation justifying this model of interaction; indeed, it implies that, in a finite slice of time for an individual with finite resources, interactions that consume time and resources never interfere. For a large interval of time, that might be reasonable – but large intervals of time conflicts with assumption that actions stimulate responses in subsequent intervals. Alternatively, we might infer that there is substantial variability in events (in terms of their duration and resource requirements) such that many interactions means small events, but this undermines the model assumption that these activities can be lumped together into a single category.

The upside of this "ideal gas" model of social activity is ease of implementation. However, we can circumscribe the appearance of this assumption in our simulation, so that some future Van der Waals can easily clean up our mess later with a more detailed interaction model.

We will use this as the probability of *any* `Bad` interaction with those groups, and that in general their interaction with these groups is binomially distributed. Therefore, if a particular plotter has k collaborators (or supervisors, or subordinates), the probability of interacting with an individual in that group, p_i, is related to the probability of interacting with the group, p_g – which is the probability in the simulation state – by:

$$(1 - p_i)^k = 1 - p_g \rightarrow p_i = 1 - \sqrt[k]{1 - p_g}$$

References

[1] Gilad Bracha and William Cook. Mixin-based inheritance. In *ACM SIGPLAN Notices*, volume 25, pages 303–311. ACM, 1990.

[2] Kathleen M Carley. Dynamic network analysis. In *Dynamic social network modeling and analysis: Workshop summary and papers*, pages 133–145. Comittee on Human Factors, National Research Council, 2003.

[3] Typesafe Inc. Akka, 2013.

[4] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, Citeseer, 2004.

A Code for Base Simulation Actor

B Messages to Construct Salafi Network

```
// ...TODO figure out these messages
val cabal = Set(``A'',``B'',``C'');
cabal foreach { _ ! Radicalize(...) }
cabal foreach { _ ! Collaboration(..., cabal) }
```