Q1) How does (mathematically) formal specification (just the specification, not the model checking) compare to property-based specification and traditional unit testing? What sorts of things seem easier to do well in each? You might consider things like ease of defining or building data structures, the precision of the specification, learning curves, or other aspects (but you don't have to necessarily talk about each of those or only those).

Ans)

Formal specifications are basically used to correctly implement the systems by defining a system in mathematical specifications. However, Property-based and traditional unit testing only helps in testing the system, not in implementing especially during the design phase.

While implementing property-based and unit testing is very similar and easy in some languages, formal specifications are very different so there is a bit more of a learning curve. This becomes more apparent in the programs that are difficult to be represented in mathematics, unit-testing becomes the only feasible option.

With studying a few formal specifications tools in this class so far, no formal specifications have given a measure of code coverage. More importantly, the completeness of a formal specification can be argued and there is no automated measure like in unit-testing to test it out.

Formal specifications actually work better with the unit testing. This is true because formal specifications can actually help developers in identifying test cases. Property-based testing would be redundant in the presence of formal specifications and unit testing.

Defining large data structures are easier in property-based testing and traditional unit testing than that in the formal specifications. One main reason for this is the limited number of structures available. On the other hand, we can even define custom data structures in the unit-testing.

Recursions methods are a very important part of our programs. Writing recursion methods are easier in property-based testing or unit-testing than in formal specifications.

Testing the efficiency of our systems is not possible with just formal specifications and property-based testing, but that can be possible with unit-testing.

Practically for developers, property-based testing and unit-testing meld very well with software development techniques like Agile whereas formal specifications still have not found space in these methodologies.

It is very hard to create tests for multi-state systems in unit-testing and property-based tests, but the formal specification can easily define state properties.

Q2) How would you pursue (traditional) testing of liveness properties (i.e., always eventually P, eventually always P...)? This isn't meant to be a trick question, so to be clear, this isn't possible in general (an execution that violates a liveness property is infinitely long, so you can't write an assertion at the end). But how would you approach the problem of building confidence in one of these liveness properties being true of a piece of software using traditional techniques (or property-based)?
ANS)

Using Traditional Unit testing:

Running the complete system, and checking liveness properties in different states would not be possible in unit testing so testing states of various components of our system is possible to check for liveness properties. One approach to do so is by building a unit test model such as put all the liveness properties of a state in a model and test it out for <X> execution time. Although, it is difficult to find the <X> time in which our properties should pass, otherwise we can safely say they failed. To make this approach better, we can specify different time brackets such as under 5 seconds, 10 seconds, 50 seconds and so on. The last bracket would result in termination with properties passing under that time considered to be passed, and properties not passed under that time would be considered failed.

Using Property-based testing:

To build confidence for liveness properties using property-based testing, one approach is to define and write general properties for a state. After that, as our system goes through different states, check each state against these generalized properties. This approach, in theory, should replicate the purpose of liveness properties. We could done this in unit testing as well, but this would be too much of an overhead for each state in the case of unit testing. Also, if possible, we can keep track of states to eliminate the duplicate states using a HashMap. In the end, a defined final execution time would still give us a measure of how many states passed the test. Depending on the size of our system, we can build a level of confidence based on our findings using this approach.

Depending on the liveness properties, we can divide properties into properties that test the systems' output and the properties that test the systems' internals. Testing the first kind of properties using the approach described in unit testing would be easier as we don't have to define extra properties.

Q3) Several of our readings' authors have reported higher confidence making changes to real systems when they have a formal model they can query (e.g., PVS in the life-critical systems paper, or TLA+ in the Amazon paper). After this (comparatively short) experience, how do you think tools like this would affect your confidence in making changes to a system? For example, consider if you were making changes to a traffic light to add turn signals or crosswalk signals. Would having a formal model make you more comfortable changing the design, not affect your confidence, or decrease your confidence? Why?

ANS) To begin with, I agree that using formal methods gives one more confidence in making changes to real systems. The reasons for that are the following:

1) The ease of using temporal logic to check all the states certainly ensures the developers the addition of functionality without any error unless properties defined are incorrect.
2) Formal methods are defined in every stage of development including the design phase. This clarity of added concepts would increase the confidence of developers significantly.
3) In the case of life-critical systems even traffic lights, maintaining precision is very important. Keeping track of changes which affects the precision is a must for the system. Using formal models, we can simply add new changes as invariants to make sure of precision. This definitely increases confidence.
4) Adding turn signals is as simple as adding another color in our traffic light with few defined controls. Adding crosswalks would be easy as well, just by defining a variable making sure our lights don't go immediately from green to yellow. Also, testing these out is easier with the formal methods with the functionality to trace back the errors.
5) Aside from the formal specifications, one could only use informal ways to check the correctness of the added functionality. This approach is more prone to errors and difficult to backed up mathematically before the implementation phase. Even in our traffic light system, suppose we have system UML for design and implementation, it still would not provide enough confidence to make changes in our system.