Proceedings of the 2013 Winter Simulation Conference
R. Pasupathy, S.-H. Kim, A. Tolk, R. Hill, and M. E. Kuhl, eds.

TEST-DRIVEN AGENT-BASED SIMULATION DEVELOPMENT

Nicholson Collier
Jonathan Ozik

Decision and Information Sciences
Argonne National Laboratory
Argonne, IL 60649, USA

ABSTRACT

Developing a useful agent-based model and simulation typically involves acquiring knowledge of the model's domain, developing the model itself, and then translating the model into software. This process can be complex and is an iterative one where changes in domain knowledge and model requirements or specifications can cause changes in the software that in turn may require additional modeling and domain knowledge. Test-driven development is a software development technique that can help ameliorate this complexity by evolving a loosely coupled flexible design, driven by the creation of many small, automated unit tests. When the focus shifts to writing small tests that exercise the simulations behavior, the larger problem of translating a conceptual model into working code is decomposed into a series of much smaller, more manageable and highly focused translations. This paper explores the application of this technique to agent-based simulation development with examples from Repast Simphony, ReLogo and Repast HPC.

1 INTRODUCTION

Developing a useful and useable agent-based simulation typically involves acquiring sufficient knowledge of the model's domain, developing the model itself, often while grounding it in more abstract theories of the domain, and then translating the model into software. The work itself can be complex and is rarely a waterfall type process where each step is completed before the next begins. Rather, the process in an iterative one where the development of the model may require additional domain knowledge that in turn requires changes in the software. Furthermore, model specifications and requirements may change also requiring changes in the software. Test-driven development (TDD) is a general software development technique that can help ameliorate this complexity. In TDD the development of the application code is driven by the creation of many small automated unit tests each of which exercise some small part of the larger application's behavior. An aspect of the extreme-programming movement that has gained popularity since the late 1990s, the technique itself gained currency with the publication of Kent Becks seminal *Test-Driven Development by Example* in 2003 (Beck 2003). It remains an important part of agile software development methodologies (Martin 2002) such as SCRUM (Rubin 2012)

This paper explores the application of this technique to agent-based simulation development. We begin with a discussion of TDD itself, followed by an exploration of TDD in the context of Agent-based Simulation development.

2 TEST-DRIVEN DEVELOPMENT

The fundamental characteristic of TDD is that the development of the application code is driven by the creation of many small automated unit tests (Beck and Gamma 1998). Each of these small tests exercises some aspect of the application code. Focusing on many small tests helps to evolve a loosely-coupled flexible design and provides built-in verification at a low-level of granularity. These tests are written iteratively in conjunction with the code they are intended to test. TDD proceeds in the following manner.

1. Identify what to test (Think Small)
2. Write a test
3. Write the code that makes the test pass
4. Repeat

The canonical process prescribes that writing the test (step 2) occurs before the actual code it is intended to test (step 3). This places the focus on the test and is intended to ensure that the "code that makes the test pass", that is, the application code, is granular enough to be testable. The emphasis is solely on the behavior to test. In addition, Beck (Beck 2003) writes: "when we write a test, we imagine the perfect interface for our operation. We are telling ourselves a story about how the operation will look from the outside." That story constitutes a kind of mini-specification of how to implement the required code.

In our own work, we have sometimes switched steps 2 and 3. However, in that case it is exceedingly important to write the model code with testing in mind and thus the admonition to "Think Small" applies equally well to the model code. The loop from step 3 back to step 1 is motivated by the obvious necessity that a single unit test and the software code that makes it pass do not constitute the entire application. Additional tests and code have to be written. More importantly though, "Repeat" also refers to running existing tests as additional functionality is added to the application and cleaning these tests up or possibly eliminating them entirely. Beck (Beck 2003) calls this latter implication of "Repeat": "making it right." In this way, the tests and the code evolve together and the programmer has confidence that the code is behaving as expected. This cycle of steps 1 through 4 and back again is intended to be executed quickly. Testing frameworks such as JUnit (Junit 2013) and the Boost Test Library (Rozental 2013) allow the tests to be easily executed individually or in batches, providing the developer with very quick feedback.

The starting point of a TDD cycle is the identification of some small concrete result and the behavior necessary to produce that result. As an example, lets assume the development of a epidemiological simulation that models the transmission of a disease among a given population. Persons in the population can be in one of three disease states: uncolonized, colonized, and infected. The transition from one state to another is determined by a variety of factors: the presence of colonized or infected persons, the type of activity a person is performing in the presence of such persons and a set of numeric model parameters governing the probability of transmission. Ultimately, in order to explore the spread of the disease, the model simulates a population of persons performing hourly activities during which they come into contact with other persons.

Given this simple description (and of course the real specifications would certainly be more detailed), we could start with any number of expected concrete results as test candidates. However, TDD recommends that we "think small" and this applies both to the test and the code that will make the test pass. The code that will make the test pass should not include extraneous details and untested dependencies. The transition algorithm is then a natural place to begin. Under certain conditions, we expect a transition to occur and under others we do not. A possible test will look like:

```
@Test
public void testTransition() {
  TransitionAlgorithm ta = new TransitionAlgorithm();
  float activityRisk = 1f, colonizedPersons = 1f, infectedPersons = 0f;
  assertEquals(DiseaseState.COLONIZED,
     ta.runUncolonized(activityRisk, colonizedPersons, infectedPersons);
}
```

Leaving aside the details of the testing framework, this test tests that the transition algorithm when run for uncolonized persons will return a result of COLONIZED when the risk associated with the activity is equal to 1 and a colonized person is present. Additional tests might test that the result is UNCOLONIZED when the risk is 0 or no colonized persons are present. Above, we mentioned that a test like this should be written first as it would focus the development on implementing the algorithm for an uncolonized to colonized transition and and guide what the interface for this might look like. Regardless of the order in which the code is written, the important point is that only the part of the TransitionAlgorithm that we are testing should be written. Anything else is extraneous at this point.

The process of building the simulation application then is one of working "inside-out." We start with small testable pieces of behavior and compose or combine these into larger pieces. For example, the TransitionAlgorithm may become part of a Person object. These larger pieces will have their own behaviors that need to be tested and these behavior will typically combine some of the behavior of their components. The components have already been tested and thus the tests of this composite behavior can focus on that without regard for the validity of the component pieces.

The benefits of working in this way can be hard to see if the the simulation is relatively simple, the domain is well-known and the scope or specifications are unlikely to change much over the course of the project. Even in this case though, TDD can still be useful in translating the conceptual model into software. Inexperienced programmers are often confused about where to start when confronted with any model. TDD can help by focusing the development on small behaviors and providing a process for coding the simulation.

The benefits are clearer when the simulation is more complex and the agent behaviors are also of a sufficient complexity to warrant this kind of "inside-out" style. In this case, the process of translating a more complex conceptual model into working code code is decomposed into a series of much smaller, more manageable, highly focused and above all testable translations. Furthermore TDD forces the developer to code in small meaningful units and these are much easier to evolve in the face of changing requirements: the "think small" dictum encourages a loose coupling between software components. More complex simulations are often developed in teams and the tests themselves provides a medium of communication between developers. The test is a kind of mini-specification that explains how to accomplish some particular task (e.g. transition a person from uncolonized to colonized).

Lastly, TDD plays an important in the verification and validation (V&V) of agent-based simulation. V&V has been discussed at length elsewhere (North, Howe, Collier, and Vos 2007, North and Macal 2007), but briefly verification is matching an implemented model to its conceptual specification. Validation is matching an implemented model to the part of the real world it ostensibly represents. Working test-first entails that each significant bit of functionality as described in a conceptual specification has a test (in fact the test is prior to the implementation of the functionality itself). This provides a very high degree of low-level verification that the code is doing what it is intended to do.

3 AGENT-BASED SIMULATION AND TDD

Agent-based modeling and simulation (ABMS) is a method of computing the potential system-level consequences of the behaviors of sets of individuals (North and Macal 2007). ABMS allows modelers to specify each agent's individual behavioral rules; to describe the circumstances or topology in which the individuals act; and then to execute the rules to determine possible system-level results. Agents themselves are individually identifiable components that usually represent decision makers at some level. Coding an ABMS consists of implementing these individual behavioral rules, the circumstances in which the individuals act, and a mechanism for executing the rules. A variety of ABMS toolkits exist that simplify this process, enabling the developer to focus more on the model-specific individual behavior and less on topology and rule execution.

3.1 Agent Behavior and TDD

We saw above in the transition algorithm example, how TDD works well as a method for developing the implementation of agent-based behaviors. The ABMS emphasis on individual behavior fits well with TDD. The process consists of writing tests for core or elemental functionality and then building up broader agent behavior from that functionality. For example, if we begin with the transition algorithm, we can then compose our agent behavior on top of it. Exploring further the specifications of our disease model, we know persons perform activities associated with individual places (workplace type activities in the workplace, home type activities in the home and so on). Certain activities and thus certain places carry a different risks of transmission. The type of activities performed in the home entail a higher degree of contact between persons than the workspace and thus the risk of transmission between persons is higher. An initial unit test that incorporates places and persons in transmission might then look like:

```
@Test
public void testPersonTransmission() {
   Workplace place = new Workplace();
   Person person = new Person();
   person.setStatus(UNCOLONIZED);
   person.setPlace(place);
   person.runTransmission();
   assertEquals(COLONIZED, person.getStatus());
}
```

Recall that unit test is a kind of story, a mini-specification, in this case a story about the elements in our simulation cooperating to implement the place specialized disease transmission. However, once we try to write the code based on this mini-specification, that is, the code that makes the test pass, the flaws in this specification become apparent. The intention is to build upon previously tested code. In our case, we are building on the tested transition algorithm. It requires an infected and colonized person count. In the implementation implied by the test above, there is no way to compute such counts. There is no collection or grouping of persons such that a count of the infected and colonized persons in that group can be tallied.

A new unit test that implies a more compatible specification and replaces the previous one might then look like:

```
@Test
public void testPlaceTransmission() {
   Person infected = new Person();
   infected.setStatus(INFECTED);
   Workplace place = new Workplace();
   place.addPerson(infected);

   for (int i = 0; i < 10; ++i) {
      place.addPerson(new Person());
   }
   place.runTransmission();

   for (Person person : place.persons()) {
      assertEquals(COLONIZED, person.getStatus());
   }
}
```

Here we create a single infected person and add it to the place. Ten other non-infected persons are also added. We then run the transmission on the place, and check that all the persons in that place have become colonized.

This example is perhaps a bit facile, but the intention is to demonstrate how tests can help define a specification for larger agent behavior, a specification that builds on those already implemented in response to previous tests. In this case, the first attempt was a poor specification, but that is precisely the advantage of TDD. The test reveals without any extraneous distractions that it was a poor way for the place and person components of the simulation to interact. It can be quickly scrapped and a better test created. The implementation of the model evolves in an iterative manner building upon what has come before, guided by the unit tests.

3.2 Agent Behavior, Toolkit Components and TDD

The examples we have seen so far have focused solely on the agent behavior and have not included any components from ABMS toolkits such as those that represent topology or facilitate rule execution. Agent behavior often requires such functionality and thus its crucial that these components can be included as part of tests. The discussion below will feature the Repast suite of ABMS tools. Repast Simphony provides a pure Java ABMS platform and dialect of Logo, called ReLogo as well as the ability to define agent behavior with state charts and system dynamics diagrams. More on Repast Simphony can be found in North et al. (2013). Repast HPC is a parallel distributed C++ implementation of Repast Simphony for Java and to a lesser extent of ReLogo. It attempts to preserve the salient features of Repast Simphony for Java and provide Logo-like components similar to ReLogo, while adapting them for parallel computation in C++. More on Repast HPC can be found in Collier and North (2012).

In order to properly apply TDD when agent behavior tests require toolkit infrastructure, the toolkit itself must supply its functionality as more-or-less loosely coupled components that can be used in tests. If the toolkit components are tightly coupled to the entire runtime infrastructure, it is difficult to achieve the desired level of granularity and testing can only be performed by running the model in the toolkits full runtime. In order to integrate toolkit components into model behavior tests, we must be able to create just enough of the toolkit infrastructure in order to run the tests.

Continuing with the disease transmission example, we have an additional specification to test: persons, once colonized, will become infected after fourteen days. Typically, a simulation toolkit provides a mechanism for simulating the passage of time. In the Repast Simphony and Repast HPC case, the Schedule component performs this function. Actions are placed on the Schedule with a "tick" value that specifies when that action will execute with respect to other actions. This can be used to simulate time: one tick can equal one hour and so forth. In order to write a test for persons becoming infected after fourteen days using the Repast Simphony toolkits then we need to use the Schedule component.

```
@Test
public void testColonizedToInfected() {
    Schedule schedule = new Schedule();
    RunEnvironment.init(schedule, null, null, false);
    Context context = new DefaultContext();
    RunState.init().setMasterContext(context);

    Person p = createPerson(schedule, context);

    assertEquals(COLONIZE, p.getStatus());
    for (int i = 0; i < 14; ++i) {
      schedule.execute();
    }
    assertEquals(INFECTED, p.getStatus());
}
```

The intention here is that we create enough of the Repast Simphony scheduling infrastructure to run the person's behavior for fourteen simulated days, at the end of which, we expect the person to have transitioned from colonized to infected. The `createPerson` method mentioned above creates a colonized Person and schedules that Person's daily behavior on the Schedule object. We can then execute the schedule fourteen times in order to run fourteen days of simulated behavior. The simulation infrastructure components will vary among toolkits, but the important point here is that the code that makes the test pass is the same code that is used in the model. In this case, the Person object would presumably check the Schedule object for the current time and transition accordingly. The only difference is in how the simulation infrastructure, such as the Schedule and other rule executing type components are initialized for use.

One of the advantages of using toolkits like Repast Simphony and Repast HPC is that they allow the user to focus on model rather than infrastructure concerns. This benefit is seen most clearly in Repast Simphony's ReLogo and Repast HPC's Logo-like components. Logo (Logo Foundation 2013) is a widely used educational programming language commonly found in K-12 classes. For many users it is easier to conceive and design a model using the Logo paradigm. Logo provides the basis for several ABMS platforms, most notably NetLogo (Wilensky 1999) and ReLogo (Ozik 2011). Repast HPC Logo attempts to leverage the Logo paradigm and ReLogo in particular to promote ease of use and further hide the complexities of implementing a parallel simulation.

Repast HPC and ReLogo use the core Logo constructs: Turtles, Patches, Links and the Observer. Turtles are the mobile agents and can move in a two-dimensional continuous space. Patches are fixed agents located at the discrete lattice points of this continuous space, one patch per point. Each patch covers an area of 1 square unit around its location. In this way, turtles can be considered on or off particular patches and many Logo-based models are written using the concept of a local patch neighborhood as their central mode of interaction. Links are just that, network links that can be formed between two turtles. The Observer provides overall model management: initializing the simulation and by default scheduling a user-implemented 'go' method to execute every iteration of the simulation. The user only needs to implement this 'go' method, the turtle and optionally the patch behaviors.

Turtle-based user defined agents are themselves created through the Observer and this can pose complications for TDD. In order to test the Turtle behavior an Observer must exist and for the Observer to exist a non-trivial amount of initialization has to take place. Both the Junit (Junit 2013) and the Boost Test (Rozental 2013) frameworks provide some help in this respect, allowing initialization code to be tagged as such so that it only runs during an initialization phase. Junit has the "@BeforeClass" annotation for this and Boost Test has the "BOOST_FIXTURE_TEST_SUITE" macro. In both cases the code tagged by @BeforeClass or contained within the BOOST_FIXTURE_TEST_SUITE macro will be executed before any of the tests themselves. The @BeforeClass runs once before all the tests are executed. BOOST_FIXTURE_TEST_SUITE runs once before each test. (Junit also has a @Setup annotation that functions like BOOST_FIXTURE_TEST_SUITE.) Using @BeforeClass (or @Setup) and BOOST_FIXTURE_TEST_SUITE then its possible create a test harness that makes the testing of Turtle-based agent behavior possible. In ReLogo this looks like:

```
static UserObserver observer;

@BeforeClass
public static void setUpBeforeClass() throws Exception {
    String scenarioDirString = "./test_data/test_scenario.rs"
    ScenarioUtils.setScenarioDir(new File(scenarioDirString));
    File paramsFile = new File(ScenarioUtils.getScenarioDir(),
      "parameters.xml");
    ParametersParser pp = new ParametersParser(paramsFile);
    Parameters params = pp.getParameters();
    RunEnvironment.init(new Schedule(), null, params, true);

    Context context = new DefaultContext();
    SimBuilder builder = new SimBuilder();
    context = builder.build(context);
    observer = (UserObserver) context.iterator().next();
}
```

The point here is to create an Observer with enough functionality that Turtle-based agents can be created and have their behavior tested. We begin by creating a Java File object that points to the file containing the model's parameters. From this file, we create a Parameters object that is used to initialize the RunEnvironment together with a Schedule instance. We then use a SimBuilder to create the Context. The SimBuilder will use the information in the Parameters via the RunEnvironment to create an Observer and place it in the Context. (UserObserver is an implementation of Observer). Once the Observer has been created we can then use it to create turtle-based agents coded by the user and test their behavior in subsequent tests.

In Repast HPC, the test initialization for a Logo-like Observer looks like:

```
class ObserverSetup {

public:
   repast::relogo::Observer* obs;

   ObserverSetup() {
      repast::relogo::WorldDefinition def(0, 0, 10, 10, true, 0);
      repast::relogo::WorldCreator creator(repast::RepastProcess::
         instance()->communicator());
      std::vector<int> vec(2, 1);
      obs = creator.createWorld<MyObserver,
         repast::relogo::Patch>(def, vec);
      repast::Properties props;
      obs->_setup(props);
   }

   ~ObserverSetup() {
      delete obs;
   }
};

BOOST_FIXTURE_TEST_SUITE(transmission_test, ObserverSetup);
```

The BOOST_FIXTURE_TEST_SUITE macro takes two arguments. One is the name of the test suite, that is, a named collection of tests, and the name of a C++ class that will run before each test in the suite is executed. In this RepastHPC case, we begin by creating a WorldDefintion that defines the salient features of the "world" in which the agents will operate, such as ithe minimum and maximum coordinates, whether the world is wrapped into a torus and the size of the buffer between the distributed world partitions. We then create a WorldCreator object that functions much as the SimBuilder does in the ReLogo case. The WorldCreator createWorld method is passed to the Observer and Patch types as template arguments and the WorldDefinition and a vector of ints are passed as method arguments. The vector defines the number of partitions over which the space will be distributed along the X and Y axes. Lastly, the Observer created by the createWorld method call is setup using a Properties object. In this case, we use an empty Properties object as there are no model specific parameters or properties to pass to the Observer. Once the Observer has been setup, the "obs" pointer can be used in tests to create turtle-based agents and then test their behavior.

These two examples, ReLogo and Repast HPC, are obviously specific to the Repast Simphony toolkit. However, the larger point should be applicable to any ABMS toolkit. Much agent behavior involves some interaction with the ABMS toolkit with which it is developed. If it is possible to initialize the toolkit such these components are easily available to test code, then a deeper layer of TDD for ABMS becomes possible.

4 CONCLUSION

In this paper, we have discussed TDD in relation to ABMS development. TDD promotes an "inside-out" development style that ameliorates the complexity of turning a conceptual model into working code. The translation of the model into code is decomposed into a series of much smaller, more manageable, highly focused and above all testable translations. Furthermore, these unit tests provide much needed flexibility in the face of changing project requirements, model designs and specifications. In the context of ABMS development, we have demonstrated how TDD can be used to implement agents' behavior when that behavior is both independent from and dependent on ABMS toolkit components. Lastly, TDD has proved exceedingly valuable in our experience, both in the actual development of the Repast Simphony ABMS toolkit and the development of models such as the ABMS Cardiovascular disease model (Graziano et al. 2013).

ACKNOWLEDGMENTS

This work is supported by the U.S. Department of Energy, Basic Energy Sciences, Office of Science, under contract #DE-AC02-06CH11357.

REFERENCES

Beck, K. 2003. *Test Driven Development: By Example*. Boston, Massachusetts: Addison-Wesley.

Beck, K., and E. Gamma. 1998. "Test Infected: Programmers Love Writing Tests". *Java Report* 3 (7): 37–50.

Collier, N., and M. North. 2012. "Parallel agent-based simulation with Repast for High Performance Computing". *SIMULATION* November.

Graziano, D., M. North, C. Sarawate, N. Collier, S. Sharan, D. Kruzikas, C. Macal, and M. Higashi. 2013. "Using agent-based modeling and simulation (ABMS) to bridge healthcare policy and private sector investment in emerging markets: case of cardiovascular disease (CVD) in Indian". In *Proceedings of the 4th Annual Consortium of Universities for Global Health*. Washington DC: Consortium of Universities for Global Health.

Junit 2013. "Junit". Accessed May 17, 2013. http://junit.org.

Logo Foundation 2013. "Logo". Accessed May 17, 2013. http://el.media.mit.edu/logo-foundation.

Martin, R. C. 2002. *Agile Software Development, Principles, Patterns, and Practices*. New York: Prentice Hall.

North, M., N. Collier, J. Ozik, E. Tatara, M. Altaweel, C. Macal, M. Bragen, and P. Sydelko. 2013. "Complex Adaptive Systems Modeling with Repast Simphony". *Complex Adaptive Systems Modeling* 1 (3).

North, M. J., T. R. Howe, N. T. Collier, and R. Vos. 2007. "A Declarative Model Assembly Infrastructure for Verification and Validation". In *Advancing Social Simulation: The First World Congress*, edited by S. Takahashi, D. Sallach, and J. Rouchier. New York: Springer Verlag.

North, M. J., and C. M. Macal. 2007. *Managing Business Complexity: Discovering Strategic Solutions with Agent-Based Modeling and Simulation*. New York: Oxford University Press.

Ozik, J. 2011. *ReLogo Getting Started Guide*. Repast Development Team. http://repast.sourceforge.net/docs.html.

Gennadiy Rozental 2013. "Boost Test". Accessed May 17, 2013. http://www.boost.org/doc/libs/1_53_0/libs/test/doc/html/index.html.

Rubin, K. S. 2012. *Essential Scrum: A Practical Guide to the Most Popular Agile Process*. New York: Addison-Wesely.

Wilensky, U. 1999. "NetLogo". http://ccl.northwestern.edu/netlogo.

AUTHOR BIOGRAPHIES

Nicholson Collier Ph.D., is a software engineer at the Center for Complex Adaptive Agent Systems Simulation within the Decision and Information Sciences Division of Argonne National Laboratory and a staff member at the Computation Institute at the University of Chicago. His email address is ncollier@anl.gov.

Jonathan Ozik Ph.D., is computational scientist at the Center for Complex Adaptive Agent Systems Simulation within the Decision and Information Sciences Division of Argonne National Laboratory and a Fellow at the Computation Institute at the University of Chicago. His email address is jozik@anl.gov.