Towards the Automatic Identification of Faulty Multi-Agent Based Simulation Runs Using MASTER

Chris J. Wright, Phil McMinn and Julio Gallardo

University of Sheffield, Department of Computer Science,
Regent Court, 211 Portobello, S1 4DP, UK.

Abstract. Testing a multi-agent based model is a tedious process that involves generating very many simulation runs, for example as a result of a parameter sweep. In practice, each simulation run must be inspected manually to gain complete confidence that the agent-based model has been implemented correctly and is operating according to expectations. We present MASTER, a tool which aims to semi-automatically detect when a simulation run has deviated from "normal" behaviour. A simulation run is flagged as "suspicious" when certain parameters traverse normal bounds determined by the modeller. These bounds are defined in reference to a small series of actual executions of the model deemed to be correct. The operation of MASTER is presented with two case studies, the first with the well-known "flockers" model supplied with the popular MASON agent-based modelling toolkit, and the second a skin tissue model written using another toolkit—FLAME.

1 Introduction

Multi-agent based modelling and simulation is an increasingly popular form of paradigm that is helping scientists, industrialists and policy makers develop their understanding of natural systems, make forecasts, and predict the impact of potential future changes [17], [3], [4], [8]. The need for rigorous model testing and testing tools is becoming ever greater, since model errors can have potentially disastrous consequences, including financial loss [16] and incorrect scientific conclusions [5]. One barrier to the thorough testing of simulation models is the time that must be spent manually inspecting a potentially enormous number of simulation executions for potential errors, which may have been produced as the result of common verification procedures such as parameter sweeps.

This paper presents MASTER (Multi-Agent based Simulation TestER). MASTER is a testing framework that aims to semi-automatically detect "suspicious" simulation runs that may indicate a fault in the implementation of a multi-agent based model. MASTER works by observing a series of simulation runs believed by the modeller to represent the "normal" behaviour of the model. The modeller then specifies a set of assertions that place bounds on which particular simulation properties of the model may deviate from those already observed. In addition, a series of so-called “facts” about the model may also be specified—states of the simulation which should never occur. MASTER then monitors further, potentially extensive, simulation executions of the model—automatically flagging up executions that deviate from normal behaviour or violate some specified fact. The end result is a smaller set of simulation runs, flagged up as suspicious, to be further examined by the modeller.

The MASTER framework was originally written for use with the MASON agent-based modelling toolkit [12], but has since been extended for FLAME [10]. This paper describes the use of MASTER with the simple flockers model supplied with MASON. Results are also presented showing the detection of suspicious runs when the code of the flockers model is randomly mutated to introduce small faults. A further case study is presented with a real-world skin tissue model [17] written for FLAME. White noise is injected into key statistics collated from the model, the presence of which is identified by MASTER.

The contributions of this paper are therefore as follows:

1. A technique for semi-automatically identifying “suspicious” simulation runs of an agent-based model, using past simulation data and modeller annotations
2. An implementation of this technique into a tool, MASTER
3. An investigation into the capabilities of the technique with two case studies, the first with the flockers model and the second with a real-world skin tissue model.

The remainder of this paper is organized as follows. Section 2 describes our technique for identifying suspicious simulation runs for multi-agent based models, implemented into the MASTER tool. Section 3 then presents the usage of MASTER with the well-known flockers model provided with the MASON Java-based agent modelling and simulation toolkit. Section 4 then presents results when MASTER is used with a real-world skin tissue model. Section 5 then presents related work while Section 6 closes with concluding remarks and avenues for future work.

2 The Technique Implemented by MASTER

In normal software testing practice, test cases are evaluated with respect to a specification of a system. However, agents tend to perform actions in a probabilistic or non-deterministic manner, meaning that—given exactly the same circumstances—an agent may choose do something different from one simulation to the next; while the interaction of agents can give rise to complex emergent behaviours, which by their nature are unpredictable and hard to specify precisely. When a specification is not present, a system is evaluated by a software tester who has a detailed knowledge of the system’s requirements and which behaviours constitute correct or incorrect behaviour. However, the manual evaluation of long simulation runs is a time-consuming and laborious process.

For a model of any reasonable complexity, generating and evaluating all possible simulation runs is an intractable task. The approach taken by the MASTER framework is to capture data from a small set of simulation runs believed to adequately represent the principal "behaviours" of a model. Following this, a much larger number of simulations can then be run and automatically checked for similarity with those previously observed model executions. A deviation from "normal" behaviour may indicate that the simulation run has exposed a previously undetected fault in the underlying code of the model. The extent of the deviation at which a simulation run is deemed to be "suspicious" is specified by the modeller. Furthermore, the modeller can specify "hard" constraints about a model that are independent of observed model executions—e.g., an agent should never move off the bounds of the grid representing the world in which they inhabit.

The various stages involved in using MASTER are depicted in Figure 1 and can be summarised as follows:

1. *Capturing* is where information regarding "normal" operation of a model in simulation is recorded from a series of sample executions. The modeller must specify what information is to be captured.
2. *Observation Generation* is where so-called "observations" are created by relating data obtained during capturing with modeller-specified assertions that place bounds on that data. These bounds relate the degree to which certain attributes may deviate in future simulation runs from the values already observed for them.
3. *Testing* involves repeated execution of new simulation runs checking for violations of observations and additional modeller-specified facts. Violating simulations are flagged up to the modeller for further investigation.

In much the same way that a tester must write a test class in an xUnit testing framework (such as JUnit [1] for testing Java classes), MASTER requires the tester to extend a common interface to specify the types of information that needs to be captured from normal model behaviour, along with the formulation of "fact" and "observation" assertions. Unlike JUnit, however, MASTER does not require the tester to write specific scenarios in which the assertions will be tested. Instead, each assertion statement is evaluated against a set of simulation runs and evaluated to see if it holds or not. These simulation runs may be generated as a result of a parameter sweep of a model, or from random starting configurations.

MASTER is written in Java for the testing of models written using either MASON [12] or FLAME [10]. The following sections explain each step involved in using MASTER in detail, with the testing of the simple MASON agent class `SpatialAgent` shown in Figure 2. `SpatialAgent` implements MASON's `Steppable` interface, which simply involves implementing the `step` method to move the agent to a new (x, y) co-ordinate at each time step of the simulation.

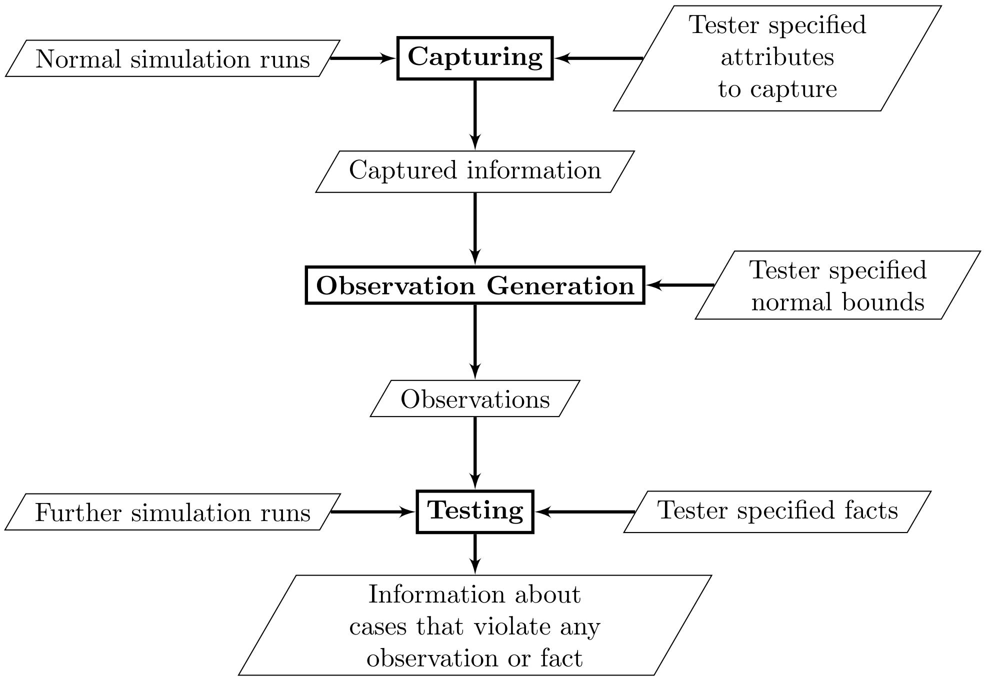

Fig. 1. The process behind MASTER. The "Capturing" stage involves recording specified information over a number of example simulation runs exhibiting "normal" behaviour. The "Observation Generation" stage is the process of combining recorded information and tester-specified bounds of deviation with respect to that information. The final "Testing" stage is where new simulation runs are evaluated against observations and "facts"—additional hard constraints specified by the tester.

```
public class SpatialAgent implements Steppable {
  private int x, y;

  public SpatialAgent(int x, int y) {
    this.x = x; this.y = y;
  }

  public void step(SimState state) {
    x = state.random.nextInt(100);
    y = state.random.nextInt(100);
  }
}
```

Fig. 2. A simple agent for demonstrating testing with MASTER. At each simulation time step, the agent moves to randomly-chosen co-ordinates.

Capturing. Recording every piece of information about a model's execution over several simulations quickly leads to a situation where an enormous quantity of data must be managed. Simulations consist of several time steps, usually involving many different agents all in different states. The *capturing* stage involves the tester writing code using the MASTER framework to track and capture specific types of data only. This reduces the quantity of data to be stored, managed, and the cost of later analysis. It also allows for richer types of information to be collated other than just raw agent state information. For example, the tester can specify that a computed value such as distance travelled by a particular agent be captured, by tracking the co-ordinates of that agent over different time steps.

In order to track specific data values in simulation using MASTER, the tester must write a *tracker* class that extends MASTER's abstract `Tracker` class. A tracker describes how raw values are captured from the state of a particular agent. An example tracker for a `SpatialAgent` can be seen in Figure 3. It is called "`XMinTracker`", and captures the minimum x co-ordinate value of the particular agent attached to the tracker at a time step of the simulation. Raw x co-ordinates are obtained using MASTER's `Reflector`, which uses Java's reflection mechanism the access the private instance variables of an agent. Information regarding what is to be accessed is specified using a "locator". A locator is simply an object that describes the sequence of method calls or instance variables required to retrieve some desired information about an agent (or set of agents).

```
public class XMinTracker extends Tracker {
  SpatialAgent agentBeingTracked;
  Locator locator;
  int min;

  public XMinTracker(SpatialAgent spatialAgent) {
    agentBeingTracked = spatialAgent;
    locator = new Locator("x");
  }

  public boolean capture() {
    int x = Reflector.retrieveInt(agentBeingTracked, locator);
    if (x < min)
      min = x;
    return true;
  }

  public Infolet getInfolet(long step) {
    return new XMinInfolet(locator, step, min);
  }
}
```

Fig. 3. Tracker code to capture the minimum x co-ordinate value of a spatial agent over the course of a simulation.

Note that the MASTER tracker code is kept entirely separate from the MASON agent code. MASTER does not require special hooks to be inserted into MASON code in order to test it. Trackers must be attached to a simulation so that data can be captured. In attaching a tracker, the tester must specify the number of time step intervals for which the data will be captured using the `capture` method. That is, an interval of 5 would lead to data being captured from `SpatialAgent` after every 5^{th} call by MASON to the agent's `step` method.

Data captured by a tracker is made available after the simulation has finished via `Infolet` objects. A specific Infolet class is implemented for each tracker to simply return the data captured and the time step that it was captured for (the creation of this class is currently a manual process, but is a step which can be automated in future). MASTER is capable of writing `Infolet` objects to a text file using the JSON (JavaScript Object Notation) common data interchange format. This is an alternative to binary serialization of objects, and allows for human-readability of information, as well as enabling the captured data to be imported easily into other tools for other types of analysis.

Observation Generation. The second stage in MASTER involves the generation of "observations" for later use in the testing phase. An observation is an assertion relating new simulation data to that already captured in the prior *capturing* phase. The assertion specifies when data from the new simulation should be regarded as "suspicious"; for example if certain values are over some defined boundary, or represent outliers (e.g., are a certain number of standard deviations from an established mean), or are found to be significantly different from those previously obtained—established using some statistical test.

Observations are written as classes that extend MASTER's `Observation` class. An example can be seen in Figure 4, `XLessThanObservation`, which asserts that all `x` co-ordinate values for `SpatialAgent`s are less than the observed minimum value from the simulations examined during capturing—denoted by the variable `observedMin`. The assertion code is found in the `check` method, which takes a "`target`"—in this case a `SpatialAgent`. The `target` variable could also refer to an entire MASON `SimStep` object, so that all agent data from a particular simulation state is accessible. As for `Infolet` objects, observations may be saved to text files in JSON form.

Testing. The *testing* phase of MASTER involves taking new simulation runs and checking each simulation step against each observation and fact. Observations and facts may be scheduled for checking at intervals rather than at every individual time step. The underlying algorithm for the testing phase can be seen in Figure 5.

If a simulation is found to violate an observation or fact, information is recorded, according to a violation handler, about the simulation and the violation that occurred. This includes the initial configuration of the model, the states of each agent present in the initial time step, any environmental parameters, and the random seed used. This allows the entire simulation to be recreated, and

```
public class XLessThanObservation extends Observation {
  int observedMin;
  Locator locator;

  public XLessThanObservation(int observedMin) {
    this.observedMin = observedMin;
    this.locator = new Locator("x");
  }

  public Result check(Object target) {
    int x = Reflector.retrieveInt(target, locator);

    if (x < observedMin)
      return Result.newSuccess();
    else
      return Result.newFailure(x);
  }
}
```

Fig. 4. Example code for an observation. The `check` method is responsible for asserting whether the data from some current simulation (passed into the method as the `target` object) is violated or not.

```
$step \leftarrow 1$
While ($step \leq maxStep$)
    Run the simulation step
    Obtain all observations and facts scheduled for $step$
    For Each observation or fact
        Check for a violation
        If violation
            Report all violation details to violation handler
        End If
    End For Each
    $step \leftarrow step + 1$
End While
```

Fig. 5. Algorithm used in the testing phase

visually inspected if necessary, to allow the tester to understand the nature of the violation and to undertake any debugging steps that may be required. MASTER provides handlers that write violation information to a file or the console, or the tester can provide their own handler that overrides the provided violation handling interface.

3 Case Study 1: Flockers Model

The "Flockers" model in MASON simulates a number of agents exhibiting co-ordinated movement with one another, as seen with natural flocks of birds or shoals of fish. Each flocker agent takes into account local spatial information when deciding which co-ordinates to move to in the next time step; including the direction and momentum of the flockers around it, the need to avoid colliding with other Flockers, coupled with a small degree of random movement. The model also includes optional "dead" flockers, that do not move, but which the live flockers try to avoid colliding with.

In order to evaluate MASTER, an experiment was performed with the Flockers model using Mutation Analysis [9]. Mutation Analysis inserts small syntactic changes to program code, which are designed to mimic typical errors made by programmers—for example, "off by one" errors, where a branching predicate in an `if` statement is changed from `x > y` to `x >= y`. A "mutant" is a piece of program code that has had exactly one syntactic change made to it. The mutant is said to be "killed" when it produces different output from the original program with the same input. The use of Mutation Analysis allows us to artificially inject errors into models, resulting in potentially faulty simulation runs. The effectiveness of MASTER can then be analysed by comparing the number of facts and observations violated by the mutated model simulations.

A special case of mutant is the *equivalent mutant*. An equivalent mutant occurs when a syntactic change cannot result in a change of output [9]. An example of an equivalent mutant is shown below. The mutation changes the relational operator of the inner-nested `if` statement from "equals" to "greater than or equals". Since `i` can never be greater than `10` as specified in the condition, there is never any difference in the behaviour of the program, despite the minor change that has been made. In general, detection of equivalent mutants is an undecidable problem.

```
if (i <= 10) {
  ...
  if (i == 10) {
    ...
  }
}
```

Original program code

```
if (i <= 10) {
  ...
  if (i >= 10) {
    ...
  }
}
```

Equivalent mutant

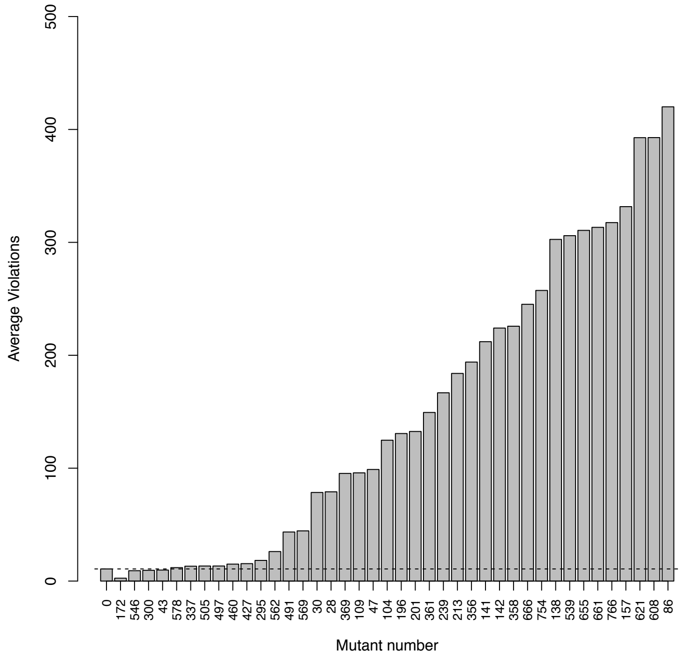

Fig. 6. Average number of observation violations for each mutant of the Flockers model. Violations (outlier values) for the original model appear as bar #0. The dotted line is plotted across the graph to show how the average number of violations for each mutant compares to the original, non-mutated model.

The original flockers model was sampled 30 times, in which data for the attributes listed in Table 1 were captured for each flocker in the tracking phase. The model was run with 40 flockers (with each flocker set to being a non-moving "dead" flocker with a probability of 0.1), for 1500 time steps. The Flockers model was mutated automatically using the MuJava tool [13], resulting in several hundred mutants, of which forty were selected at random. Each mutated model was then run again to check for observation violations. The attributes listed in Table 1 triggered a violation if they were over two standard deviations from the recorded mean for that property in the previous tracking phase. The purpose of this observation is to trap outlying behaviour of the model.

Each mutated model was run 50 times to obtain an average. The average number of property violations for each mutant can be seen in Figure 6. Each mutant is assigned a unique identification number, with the original non-mutated model assigned an ID of 0 and appearing as the left-most bar in the chart.

The average number of property violations per mutant recorded in Figure 6 correlates well with visual observations comparing original model behaviour with mutated model behaviour, as recorded in Table 2. Moving from left to right in Figure 6, the first 16 mutants up to #28—apart from #569—are recorded in Table 2 as having no visually detectable difference in behaviour (i.e., potentially

Table 1. Flocker attributes captured during tracking for the flockers model. With the exception of "distance travelled" each attribute is accessed directly from each individual flocker—i.e., from an instance variable or an accessor method of each flocker object.

Property	Description
Position	The X and Y co-ordinates of each flocker.
Momentum	The X and Y momentum values of each flocker. High momentum values encourage a flocker to keep travelling in the same direction.
Avoidance	The X and Y avoidance values of each flocker. High avoidance values encourage a flocker to keep a minimum distance from other flockers.
Cohesion	The X and Y cohesion values of each flocker. High cohesion values encourage a flocker to towards the local area containing the majority of flockers.
Consistency	The X and Y consistency values of each flocker. High consistency values encourage a flocker to move similarly to other nearby flockers.
Orientation	The orientation value (in radians) of each flocker. The orientation value represents the direction the flocker is facing.
No. of neighbours	The number of neighbours throughout the simulation that are close enough to a flocker such that the information regarding those neighbours factor into its cohesion, avoidance and consistency calculations.
Distance travelled	The last position of each flocker is stored in order for the distance travelled by each flocker to be computed and tracked.

"equivalent" mutants). The first 10 mutants up to and including mutant #427 in the graph Figure 6 show little difference in terms of observation violations (outlying statistics) when compared with the non-mutant #0, the original model.

4 Case Study 2: Skin Tissue Model

The skin tissue model [17] is written using the FLAME multi-agent based modelling and simulation environment [10], and is designed to simulate colonies of skin cells on a laboratory culture plate. The simulation begins with a few randomly-seeded individual cells, which form the epicentre of a colony. In each time step of the model, cells progress through the cell cycle and divide, producing new cells. Colonies grow outwards from the initial cell, eventually covering the entire plate. One important aspect of the model is the so-called "differentiation" of a skin cell from one type to another (e.g., to a "corneocyte" skin cell found in the upper-most layers of skin tissue). In the model, cells change type based on the distance from the centre of the skin cell colony of which they are a part. For the purposes of evaluating MASTER, a function was introduced into the model which applied a random proportion of noise to this distance property, thus introducing a source of potential simulation error into the model.

In evaluating MASTER, the model was run for 1000 time steps, with 50 runs performed for tracking and 10 repetitions with three proportional noise levels (low, medium and high). Six skin cells were initially seeded for each simulation run at random locations on the culture plate. In tracking and testing, the distance

Table 2. Visual descriptions of each simulation for each model after a mutant has been applied

Mutant	**Difference**
28	*No visual difference detectable*
30	*No visual difference detectable*
43	*No visual difference detectable*
47	Flockers show a strong preference to flying towards the right of the screen
86	Flockers gradually disappear
104	Flockers arrange into up to three evenly-spaced horizontal bands
109	Flockers tend to move in horizontally aligned formations
138	Flockers move to the left only
141	Flockers move vertically only
142	Flockers move vertically only
157	Flockers move downwards only
172	*No visual difference detectable*
196	Flockers attract one another, causing "piles" of flockers to develop
201	Flockers attract one another, causing "piles" of flockers to develop
213	Flockers move downwards only
239	Flockers do not flock
295	Flockers do not flock consistently together as normal
300	*No visual difference detectable*
337	*No visual difference detectable*
356	Flockers stabilise to move consistently along the X axis
358	Flockers stabilise to move consistently along the Y axis
361	Flockers stabilise to move consistently along the Y axis
369	Flockers do not flock, moving as individuals or pairs
427	*No visual difference detectable*
460	*No visual difference detectable*
491	*No visual difference detectable*
497	*No visual difference detectable*
505	*No visual difference detectable*
539	Flockers move downwards only, avoiding each other
546	*No visual difference detectable*
562	Flockers move mostly normally, with occasional erratic turns
569	Flockers move mostly normally, but do not form large groups moving together
578	*No visual difference detectable*
608	Flockers gradually disappear
621	Flockers gradually disappear
655	Flockers move mostly only horizontally to the right only
661	Flockers move mostly only vertically to the bottom of the screen only
666	Flockers move in normal patterns, but slowly and jerkily
754	Flockers move in almost the same direction all of the time
766	Flockers move mostly vertically only

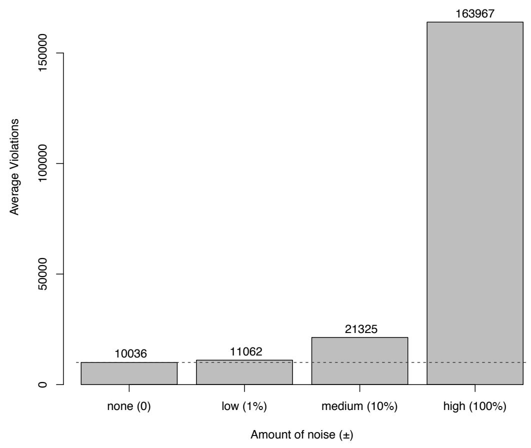

Fig. 7. Average number of observation violations for the skin tissue model for various levels of noise. Violations (outlier values) for the original model appear as noise level 0 ("none"). The dotted line is plotted across the graph to show how the average number of violations for each model with noise compares to the original model without noise.

attribute before a cell makes its first differentiation into another skin cell type is monitored. Figure 7 shows the number of observation violations that occurred when the distance attribute strayed over two standard deviations from the mean found for the property during the tracking step. Low levels of noise (up to 1% of proportional noise applied to the attribute during testing) result in little difference from the original model without noise, but many violations occur with higher levels of noise (± 10-100%) and as such are easily detected by MASTER.

5 Related Work

MASTER is a tool for testing the results of whole simulation runs of multi-agent based models. While there has been work on testing agent-based systems, there has been little work that specifically addresses testing of agents designed for simulation.

SUnit, for example, is an existing testing framework for multi-agent systems (MAS), based heavily upon the JUnit framework, that provides an approach for the testing of individual agent behaviour. JAT [6] is similar to SUnit, but uses "mock" agents to send messages to the "agents under test", and then compares the resulting replies against the expected responses. Nguyen et al. [15] propose "eCat", which follows a "goal-oriented" approach in which *means-end* scenarios are described, such that a series of actions (e.g., message passes between agents) should result in a particular goal being achieved, for example a final message containing a given piece of information. Zhang et al. [18] make use of design artefacts, in this case from the *Prometheus* design process, to generate the test data. The data takes the form of "test plans" which describe the various conditions required to evoke a particular behaviour and the predicted outcomes. This, along with a focus on message passing style agents, leads to an "agent-centric" testing approach, where the behaviour of each agent is examined in isolation from other agents and their environment, ensuring that the agent responds correctly to particular messages and percept information.

VOMAS, proposed by Niazi et al. [14], is one tool for validating and verifying multi-agent based simulations. Agents are grouped together by an "overlay" agent. The agents of this overlay are then able to define constraints describing unusual behaviour, and report violations of these if they occur. This validation may relate to both spatial data, i.e. the exact positioning or relative distance of the agents in the simulation under test, and non-spatial data, such as the edges in a graph of connected agents in a social simulation. However, it is not clear how the constrains for the overlay agents are derived, other than from subject matter experts, who provide these during the design of the overlay MAS. Rather than relying on such experts, the MASTER approach attempts to determine the boundaries for these normal values semi-automatically based upon human-approved runs—using some user-specified tolerance outlier formula. MASTER then allows the use of "facts" to allow such expert knowledge to also be incorporated—if there are known domain-specific constraints.

MASTER differentiates itself from the discussed works by both allowing the user to determine the appropriate level of testing, such as applying agent-specific or simulation wide as facts or observations, and reducing reliance on subject area experts, by determining "normal" boundaries from user-approved runs.

6 Conclusions and Future Work

This paper has described a technique for semi-automatically detecting anomalous behaviour in simulations of multi-agent based models. This technique has been implemented into a prototype tool called MASTER. MASTER involves capturing sample data from simulation runs confirmed by a tester to be behaving "normally". Testing of further simulation runs is then directed at comparing whether those simulations have deviated from those witnessed previously automatically. This removes some reliance on expert users, who may otherwise need to manually examine or analyse data produced from a simulation. This allows users to more thoroughly examine the behaviour of their model and ensure, for example, how variation of parameters may affect some emergent behaviour, improving the understanding of the given agent-based simulation.

Future work intends to incorporate of statistical analysis and more sophisticated anomaly detection routines, such as those provided by the libAnomaly [2] library [11], since presently with MASTER, the tester must specify a method for calculating bounds over captured data from "normal" behaviour, which quantifies the ranges to which future behaviour should be compared against. The idea behind anomaly detection systems is similar in principle to that behind MASTER—compare current system behaviour against a representation of normal behaviour. Anomaly detection has been successfully applied to detect malicious JavaScript code on websites, which could harm a user's system [7].

MASTER is available at: http://agents.group.shef.ac.uk/master/download/

Acknowledgements

This work was funded by the EPSRC grant EP/G009600—"Automated Discovery of Emergent Misbehaviour".

References

1. *JUnit.* http://www.junit.org (Accessed: April 2012).
2. *libAnomaly.* http://www.cs.ucsb.edu/ seclab/projects/libanomaly/index.html (Accessed: April 2012).
3. K. Bentley, H. Gerhardt, and P. Bates. Agent-based simulation of notch-mediated tip cell selection in angiogenic sprout initialisation. *Journal of Theoretical Biology*, 250:25–36, 2008.
4. M. Buchanan. Meltdown modelling. Could agent-based computer models prevent another financial crisis? *Nature*, 460(7256):680–682, 2009.
5. G. Chang, C. B. Roth, C. L. Reyes, O. Pornillos, Y.-J. Chen, and A. P. Chen. Retraction of: Pornillos et al. (Science 310 (5756) 1950-1953); Reyes and Chang (Science 308 (5724) 1028-1031); Chang and Roth, (Science 293 (5536) 1793-1800). *Science*, 314:1875, 2006.
6. R. Coelho, E. Cirilo, U. Kulesza, A. von Staa, A. Rashid, and C. Lucena. JAT: A Test Automation Framework for Multi-Agent Systems. *2007 IEEE International Conference on Software Maintenance*, pages 425–434, Oct. 2007.
7. M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the International World Wide Web Conference (WWW2010)*, pages 281–290. ACM Press, 2010.
8. J. Farmer and D. Foley. The economy needs agent-based modelling. *Nature*, 460(7256):685–686, 2009.
9. Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.
10. M. Kiran, P. Richmond, M. Holcombe, L. S. Chin, D. Worth, and C. Greenough. FLAME: simulating large populations of agents on parallel hardware architectures. In *Proceedings of the International Conference on Autonomous Agents and Multi-agent Systems (AAMAS 2010)*, pages 1633–1636. ACM Press, 2010.
11. C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the detection of anomalous system call arguments. In E. Snekkenes and D. Gollmann, editors, *Computer Security ESORICS 2003*, volume 2808 of *Lecture Notes in Computer Science*, pages 326–343. Springer Berlin / Heidelberg, 2003.
12. S. Luke, C. Cioffi-Revilla, L. Panait, and K. Sullivan. Mason: A new multi-agent simulation toolkit. In *Proceedings of the 2004 SwarmFest Workshop*, 2004.
13. Y.-S. Ma, J. Offutt, and Y.-R. Kwon. MuJava: An automated class mutation system. *Journal of Software Testing, Verification and Reliability*, pages 97–133, 2005.

14. A. H. Muaz Niazi and M. Kolberg. Verification and validation of agent-based simulation using the vomas approach. In *Proceedings of the Third Workshop on Multi-Agent Systems and Simulation'09 (MASS '09)*, 2009.
15. C. Nguyen and A. Perini. Automated continuous testing of multi-agent systems. *Workshop on Multi-Agent Systems*, 2007.
16. K. Simons. Model error—evaluation of various finance models. *New England Economic Review*, pages 17–28, 1997.
17. T. Sun, P. McMinn, S. Coakley, M. Holcombe, R. Smallwood, and S. MacNeil. An integrated systems biology approach to understanding the rules of keratinocyte colony formation. *Journal of the Royal Society Interface*, 4:1077–1092, 2007.
18. Z. Zhang, J. Thangarajah, and L. Padgham. Automated unit testing for agent systems. In *2nd International Working Conference on Evaluation of Novel Approaches to Software Engineering (ENASE-07)*, pages 10–18. Citeseer, 2007.