Mutation Operators for Agent-Based Models

Salem F. Adra and Phil McMinn
Department of Computer Science, The University of Sheffield
Regent Court, 211 Portobello, Sheffield, UK, S1 4DP
s.adra@sheffield.ac.uk p.mcminn@sheffield.ac.uk

***Abstract*—This short paper argues that agent-based models are an independent class of software application with their own unique properties, with the consequential need for the definition of suitable, tailored mutation operators. Testing agent-based models can be very challenging, and no established testing technique has yet been introduced for such systems. This paper discusses the application of mutation testing techniques, and mutation operators are proposed that can imitate potential programmer errors and result in faulty simulation runs of a model.**

***Keywords*-Mutation testing; agent-based modelling and simulation.**

I. Introduction

Computational models are computer programs designed to simulate complex systems; for example financial markets and natural systems such as skin tissue and insect colonies. Scientists and industrialists use computational models to help develop their understanding of the natural system being modelled, to make forecasts, and to predict the impact of some changes to the system. With this in mind, it is of high importance that the models have been properly tested. Recent scientific software errors have led to papers being retracted from *Science* [1]. Empirical work by Hatton [2] found an average of eight serious faults in every 1000 lines of C code analysed in a series of large scientific programs. In the banking sector, losses made by NatWest, Barclays and Deutsche Morgan Grenfell totalling tens of millions of pounds were blamed on decisions that involved economic model errors [3].

Agent-based modelling is an increasingly popular paradigm which has been successfully used to model a wide variety of applications and abstract systems. The agent-based approach focuses exclusively on modelling the micro-behaviours of the system's main actors (the agents). Agents have been used to model individual people in a crowd control simulation, cells in skin tissue [4], and the main players in an economy; for example banks, countries, households and firms [5]. During simulation, agents interact to produce complex macro-behaviours, so-called 'emergent' behaviours, such as the self-organisation of blood vessel membranes in response to a tumour [6], uprising of terrorist activity [7] or voting trends in election campaigns [8]. Figure I is a screenshot from a model of skin tissue from Sun, McMinn *et al.* [4], for predicting optimal conditions for maximal tissue growth; work of high relevance to scientific researchers developing skin replacement therapies for patients suffering heavy skin loss through burns or chronic disease.

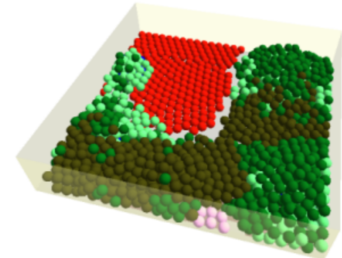

Figure 1. An agent-based model of skin tissue, with cells modelled as spheres on a virtual culture plate

Despite the power of the agent-based approach, there has been little work devoted to testing agent technologies. Mutation analysis is one way which might aid the development and comparison of suitable techniques. However, as argued in this paper, agent-based models have several different aspects that make their development unique compared to most traditional programming paradigms. In agent-based modelling, each agent is autonomous and the macro-behaviour observed in a simulation run is heavily reliant on their continuous inter-agent communication.

The contribution of this short paper is the introduction of a set of mutation operator classes for agent-based models. These operators are intended to target the type of faults that may be introduced into the coding of an agent-based model, with specific regard to the unique aspects that make up the agent-oriented style of model development. While there are many types of agent-based model and many types of framework in which to implement and execute them (for example MASON [9] or FLAME [10]), agent-based models share the same set of common 'ingredients'. With this in mind, the operators proposed are introduced in an abstract manner.

II. The ingredients of an Agent-based model

Agent-based models are a form of multi-agent system (MAS) [11] in which a system is composed from a number of autonomous interacting entities or units, referred to as *agents*. In the multi-agent system literature [12], agents have been traditionally classified as being either *deliberative*, *reactive* or hybrids of the two. A deliberative agent is usually

proactive and formulated in terms of explicit goals that the agent is trying to achieve.

Reactive agents, on the other hand, are usually not goal-oriented and only respond to environmental changes and inter-agent communication and interaction. Based on their local conditions and the conditions of their direct environment, reactive agents choose to perform certain actions based on a series of rules. These are the types of agents used in developing scientific models and running simulations.

The following formalism of an agent-based model is a modified version of that due to Kidney [13] and Denzinger [14]. An agent-based model $ABM = (A, E)$ is composed of a set of agents A in an environment E. An agent $a_i \in A$ is defined as a quadruple $a_i = (Mem, Fn, Mo, Mi)$, where Mem is a set of memory variables whose values define the agent's current state, Fn is a set of functions that an agent can execute, Mo is a set of output messages that an agent can send or broadcast, and Mi is a set of input messages that an agent can receive. The environment E is a set of global variables V which can be modified by the agents in A (or be set externally).

A particular agent $a_i \in A$, where $i = [1, ..., n]$ and $n = card(A)$ is the total number of agents, can be represented by the quadruple $(Mem(a_i), Fn(a_i), Mo(a_i), Mi(a_i))$ where $Mem(a_i)$ is the set of memory variables mem_k, where $k = [1, ..., m]$ and $m = card(Mem(a_i))$ is the number of memory variables an agent a_i has, $Fn(a_i)$ is a set of functions that agent a_i can execute, $Mo(a_i)$ is a set of output messages that a_i can send or broadcast, while $Mi(a_i)$ is set of input messages that a_i can receive. Depending on its current situation, agent a_i can choose a specific message mo_r from $Mo(a_i)$, where $r = [1, ..., l]$ and $l = card(Mo(a_i))$ is the total number of different messages in $Mo(a_i)$, to communicate with other agents. Messages in $Mo(a_i)$ are therefore used by a_i to send requests or updates to some or all other agents in A. On the other hand, every message $mi_v \in Mi(a_i)$, where $v = [1, ..., w]$ and $w = card(Mi(a_i))$, represent specific updates or requests that other agents use to communicate with a_i. At time (or iteration) t, agent a_i can then decide to execute a certain function $f_u \in Fn(a_i)$, where $u = [1, ..., z]$ and $z = card(Fn(a_i))$, based on its current situation at time t, $S(a_i, t)$: $S(a_i, t) \rightarrow f_u \in Fn(a_i)$. An agent a_i's situation at time t, $S(a_i, t)$, is defined by a_i's state at time t, $Mem(a_i, t)$, and any incoming messages $Mi(a_i, t)$ denoting a new interaction: $S(a_i, t) = Mem(a_i, t) \times Mi(a_i, t)$.

Using this formalism, the pseudocode for executing an agent-based model can be stated as in Figure 2.

for each time-step t **do**
  for each agent $a_i \in A$ **do**
    - Read any incoming messages $Mi(a_i, t)$ from agents $a_j, (j = 1..n)$ and $j \neq i$
    - Execute function(s) $f_u \in Fn(a_i)$ and update state $Mem(a_i, t)$ and E as determined by internal rules and external signals $Mi(a_i, t)$
    - Send output messages $Mo(a_i, t)$ to update or send requests to other agents a_j
  end for
end for

Figure 2. Pseudo-code describing the steps involved in running an agent-based simulation

A. Agents vs objects

A common misconception with agent-based systems is the confusion between *agents* and *objects* in object-oriented systems. While there are a lot of similarities, one large difference lies in terms of an agent's ability to be autonomous [15].

While objects and agents both encapsulate some private variables that might represent their different internal states, agents usually possess a degree of control over their state, choosing which action to perform next. An object, on the other hand, is subject to changing its state on the basis of one of its methods being called. While agents can send messages to one another, any requests are not guaranteed to be honoured, for example in scenarios where a request might conflict with the goals or local state of the agent receiving the request.

While object-oriented mutation testing operators (and more classical mutation operators) can be used for certain aspects of an agent-based model, they fall short of addressing the novel aspects of an agent-based model. Conversely, agents may not necessarily be programmed using an object-oriented language (for example the agents defined using FLAME [10]), hence features like inheritance and polymorphism do not necessarily play a role in agent-based modelling.

B. Concurrency

Similarly, concurrency may or may not cause issues for an agent-based model. This often depends on the framework being used. For example, the simulator may choose to perform the inner-most loop of Figure 2 for every agent at once, while others, for example FLAME [10], randomise the order of the agents and process them one at a time. Generally the modeller leaves such issues to the framework and does not hardwire synchronisation mechanisms into the development of a model [15]. As such, concurrent aspects of a model are not considered by the mutation operators proposed in this paper.

C. Testing Agent-based models

Galán *et al.* [16] described and classified the common errors and artefacts that can occur when developing an agent-based model. Galán *et al.* highlighted that such errors can

occur at any stage of the agent-based modelling lifecycle; for example, abstracting properties from a natural system, defining the model's technical specification or coding the model using a certain programming language. In order to detect model errors, the authors suggest some informal testing measures such as the application of the model to extreme scenarios or the re-implementation of the model using different programming languages, programming paradigms or agent-based modelling frameworks.

Another potential challenge of testing agent-based models concerns the complexity of these models. It is quite common to have a model simulating the interactions of thousands or millions of agents. Such complexity makes it very hard to understand everything going on in the system or trace back certain model behaviour to a certain agent or event.

While there has been substantial dedicated research addressing the issue of testing object-oriented systems and producing formal testing techniques for such systems, the recognition of agent-based systems as an independent set of systems with new challenges and properties - as well as the development of testing tools for these models, are topics that still need further research and investigation.

In the mutation testing community, object-oriented systems have been identified as a unique trend of software [17] [18] and several tools (e.g. MuJava [18] and Javalanche [19]) and suitable classes of mutation operators (commonly known as *class mutation*) addressing object-oriented properties such as *inheritance* and *polymorphism* were produced to address these systems. However, none of these tools or techniques are sufficient for targeting the more unique aspects of an agent-based model. As a result, the aim of this short paper is to suggest new mutation operators that are more fine-tuned for addressing the properties of agent-based models. Thus, the operators suggested in the next section can also be seen as further adaptations and refinements to previously introduced traditional, interface [20] and class mutation operators. The implementation of the suggested mutation operators are ultimately aimed at creating mutation operators that can automatically confine the scope of their corresponding mutations to the unique properties of agent-based models (i.e. agents' memory variables, communications, different functionality which can be linked to situatedness and the agents' environment).

III. MUTATION OPERATORS FOR AGENT-BASED MODELS

In this section, some mutation operators that specifically address the unique aspects of an agent-based model are proposed. The mutation operators suggested are meant to deal with potential programming, or modelling syntactical errors that can affect the behaviour and thus the reliability of a model. The intention is that these operators be combined with existing operators from the procedural and object-oriented paradigm to obtain a complete set of mutation operators for the model in hand. The mutation operators proposed in this paper have a scope which focuses on mutating the essential aspects of an agent-based model, *i.e.*:

- Agents' communication (Mo and Mi);
- Agents' memory variables (Mem);
- Agents' function executions (Fn), and
- the environment (E)

To better illustrate the suggested mutation operators, an example of an agent-based model, abm, with a total number of agents $n = 10$, is deployed (i.e. $A = a_1, a_2, \dots, a_{10}$). For simplicity, and without any loss of generality, abm is considered to be composed of homogenous agents possessing the same number of memory variables m, the same number of agent functions z and the same number of output and input messages l and w. For this example, the following values are considered: $m = 5, z = 5, l = 4$ and $w = 4$. Hence, in abm, an agent $a_i \in A$ is defined as ($\{mem_1, \dots, mem_5\}$, $\{f_1, \dots, f_5\}$, $\{mo_1, \dots, mo_4\}$, $\{mi_1, \dots, mi_4\}$).

A. Mutation of agent communication

Miscommunication

Synopsis: Mutate the set of recipient agents $R \subset A$ that are meant to receive a certain message $mo_r \in Mo(a_i)$ from agent a_i: $MisMutOp(a_i \xrightarrow{mo_r} R) = a_i \xrightarrow{mo_r} R'$, where R and R' are subsets of A and $R \neq R'$.

Examples: Examples include mutating the identity, type(s), location(s) or location ranges of an intended recipient agent. Miscommunication mutations that might be introduced by such operator are shown in Table 1 where a_i, a_j, a_{j1}, a_{j2}, a_{j3}, and $a_{j4} \in A$, $a_j \neq a_{j'} \in A$, $mo_r \in \{mo_1, mo_2, mo_3, mo_4\}$ and the underlined cells highlight the mutated values.

Table I
EXAMPLES OF MISCOMMUNICATION MUTATIONS (S = SENDER, M = MESSAGE AND R = RECIPIENT AGENT(S))

Original Message			Mutated Message		
S	**M**	**R**	**S**	**M**	**R**
a_i	mo_r	a_j	a_i	mo_r	$\underline{a_{j'}}$
a_i	mo_r	a_{j1}, a_{j2}, a_{j3}	a_i	mo_r	$\underline{a_{j1}, a_{j2}}$
a_i	mo_r	a_{j1}, a_{j2}, a_{j3}	a_i	mo_r	$\underline{a_{j1}, a_{j2}, a_{j3}, a_{j4}}$

Rather than sending messages to specific agents, some models incorporate agents that 'broadcast' messages to other agents in the vicinity. The set of recipient agents R is therefore the set of agents within a certain radius of the broadcasting agent. In such circumstances a concrete implementation of the miscommunication operator would be for messages to be received by agents outside of this range, as illustrated in Figure 3.

Corrupt message

Synopsis: Mutate a certain message mo_r that an agent a_i can send to a set of recipient agents $R \subset A$:

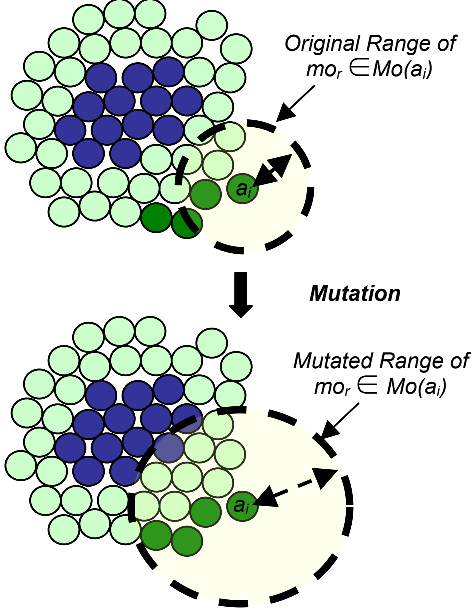

Figure 3. Mutating the range of message mo_r for agent a_i

$CorrupMutOp(a_i \xrightarrow{mo_r} R) = a_i \xrightarrow{mo_r\prime} R$, where $mo_r\prime$ is a mutated version of mo_r. Note that the suggested mutation operators affecting agent communication can be equally used to mutate input messages that an agent a_i can receive since input messages represent output messages sent by other agents $\in A$.

Examples: Examples of mutations mutating a communication message might include: (1) skipping some message data, (2) mutating the type of a message variable, or (3) mutating the choice of message to be sent in a certain situation. These three mutation cases result in sending the wrong choice of communication message, or sending a corrupted message, and are illustrated in Table 2 where $mo_r, mo_{r1}, mo_{r2} \in \{mo_1, \ldots, mo_4\}$, $mi_v, mi_{v1}, mi_{v2} \in \{mi_1, \ldots, mi_4\}$, $mo_r\prime$ and $mi_v\prime$ denote mutated versions of the original messages mo_r and mi_v respectively.

Table II
EXAMPLES OF WRONG MESSAGE OR WRONG MESSAGE CONTENT MUTATIONS (S= SENDER, M=MESSAGE AND R = RECIPIENT AGENT(S))

Original Message			Mutated Message		
S	**M**	**R**	**S**	**M**	**R**
a_i	mo_r	a_j	a_i	$\underline{mo_r\prime}$	a_j
a_i	mo_{r1}	a_j	a_i	$\underline{mo_{r2}}$	a_j
a_j	mi_v	a_i	a_j	$\underline{mi_v\prime}$	a_i
a_j	mi_{v1}	a_i	a_j	$\underline{mi_{v2}}$	a_i

Corrupt message mutations can be realised for example by skipping a z coordinate information from a location update message or changing the type of a message content variable (e.g. from a `double` to an `int`). On the other hand, a mutation operator mutating the choice of message to be sent by a certain agent a_i can be realised for example when an agent a_i sends a request for agent a_j to *sell stock* instead of *buy stock*:

B. Mutation of an Agent's Memory

Synopsis: Mutate the memory structure or content of an agent a_i at time (or iteration) t: $MemMutOp(Mem(a_i, t)) = Mem(a_i, t)'$, where $Mem(a_i, t)' \neq Mem(a_i, t)$.

Examples: Examples of agents' memory mutations might include: (1)mutating the type of memory variable (e.g. from a `double` to an `int`), or (2) mutating the values of certain constants held in memory, e.g. an enumerated agent type such as $1 = buyeragent, 2 = selleragent$ etc.

These mutations are illustrated in Table 3 where the mutation illustrates a particular memory variable being mutated ($mem_{j1} \rightarrow mem_{j1}\prime$).

Table III
EXAMPLES OF AGENT MEMORY MUTATIONS

Original Memory		Mutated Memory	
Agent	$Mem(a_i, t)$	**Agent**	$Mem(a_i, t)$
a_i	$\{mem_{j1}, \ldots, mem_{j5}\}$	a_i	$\underline{\{mem_{j1}\prime, \ldots, mem_{j5}\}}$

C. Mutating Agents' Function Execution

Synopsis: Mutate agent a_i functionality at a certain time (or iteration) t: $FnExecMutOp(S(a_i, t) \rightarrow f_u) = S(a_i, t) \rightarrow f'_u$, where f'_u is a mutated version of f_u and $f_u \in Fn(a_i)$.

Examples: The functionality of an agent a_i which is in a certain state $Mem(a_i, t)$ and receiving a certain input message $Mi(a_i, t)$ can be mutated by mutating the choice of function to be executed in such situation. Furthermore, the agent a_i's function can be mutated itself by applying traditional mutation (or if appropriate, class mutation) operators on specific lines of code defining it. Table 4, where $f_{j1}, f_{j2} \in Fn(a_i)$, $S(a_i, t)$ present a certain situation that agent a_i can be in at time t, and $f_{j1}\prime$ presents a mutated version of function f_{j1}, illustrates these kinds of agent function mutations.

Table IV
EXAMPLES OF AGENT FUNCTION MUTATIONS

Original Function			Mutated Function		
Agent	**Situation**	**Function**	**Agent**	**Situation**	**Function**
a_i	$S(a_i, t)$	f_{j1}	a_i	$S(a_i, t)$	$\underline{f_{j1}\prime}$
a_i	$S(a_i, t)$	f_{j1}	a_i	$S(a_i, t)$	$\underline{f_{j2}}$

An example showing a mutation affecting the choice of function to be executed by a certain agent a_i in a certain situation can be depicted in the following: **IF** *Stock Price is up* **THEN** *sell* → **IF** *Stock Price is up* **THEN** *buy*.

D. Mutating the Environment

Synopsis: Mutate the environment E of an agent-based model: $EnvironMutOp(E) = E'$, where E' is a mutated version of E.

Examples: Mutating the environment E can be realised for example by mutating the types or values of any environmental constant V. Table 5 illustrates such environmental mutations using an agent-based model example where the environment E is composed of 6 environmental global variables $\{V_1, \dots, V_6\}$. In Table 5, the mutation illustrates a particular environmental variable being mutated ($V_1 \rightarrow V_1\prime$).

Table V
EXAMPLES OF ENVIRONMENT MUTATIONS

Original Environment		**Mutated Environment**	
	Environment Variables		**Environment Variables**
E	$\{V_1, \dots, V_6\}$	E	$\{\underline{V_1', \dots, V_6}\}$

An example of environment mutations can be depicted for example when a certain environmental constant V defining the size of a grid containing all interacting agents or defining a concentration of a certain soluble factor which affects the agents' functionality is mutated (e.g. $gridsize = 100 \rightarrow gridsize = 80$ or $calciumconcentration = 0.1 \rightarrow calciumconcentration = 1.0$).

IV. Related work

Testing agent systems with *goal-oriented* agents have been investigated and has been attracting increasing attention. The approach to testing such agents has mainly revolved around the idea of injecting a mock or faulty agent in the system in order to assess how the other agents being tested would interact with it. In [13] and [14] for example, the authors injected mock agents to test multi-agent systems where the agents' goal was to rescue survivors in a virtual world simulating a city struck by an earthquake. The agents hence had clear objectives, and the mock agent was designed to conflict with the agents being tested to assess their behaviour in unexpected scenarios.

However, there has been very little work targeting the testing of the *reactive*-type of agents found in agent-based models. Merelli and Young [21] for example, suggested the injection of mutations into a biological agent-based model to assess the model's fidelity. Merelli and Young's approach was non-generic and consisted of injecting an agent-based model with domain specific mutations (e.g. calcium concentration $[Ca^{++}] = 0.09\ mM$ instead of the physiological concentration $[Ca^{++}] = 2\ mM$) that are known to cause certain expected, mutated, behaviour in the modelled system. The behaviour of the mutated model was then compared with the behaviour of the mutated target system to validate the functionality of the agent-based model. Shan and Zhu [22] on the other hand introduced a more generic mutation testing tool that is specifically designed to test graphical software applications which are used to model agent-based systems. Their suggested technique was however concerned with data mutation, *i.e.* mutating the input data to a certain agent-based model design rather then the model itself, and the automatic generation of test cases.

V. Conclusions and Future Work

This short paper argued that agent-based models are an independent class of software applications with unique properties and testing challenges. A simple formal definition for an agent-based model was defined and some design ideas for mutation operators which specifically address this class of software applications were proposed. The suggested mutation operators may help establish formal testing techniques and hence increase the reliability and correctness of such complex models. In addition to their testing purposes, such mutation operators may also help shed more light on the functionality and abstraction of some models, and may usefully steer the direction of further scientific experiments and investigations.

Future work will include the implementation of such mutation operators and their potential investigation in popular agent-based frameworks such as MASON [9] or FLAME [10]. Introducing mutations into an agent-based models can take place at the template level which defines the structure and functionality of all agents (or at least a certain type of agents) or at the agent level (i.e. mutating the structure and/or functionality of a specific agent or subset of agents). Investigating these two mutation levels is an interesting and important aspect that will be investigated in future work. While mutating at the template level seems more straightforward, mutating at the agent level might be more difficult to detect and more representative of agent-based models faults that might be related to situatedness. Adopting an agent-level approach for mutation testing involves some challenges such as investigating when and which agent to be mutated.

On the other hand, in order to kill mutants, agent-based models need to be simulated for a certain amount of time. This raises some challenging issues such as: how long should a model be simulated for, and how to differentiate between desired (yet previously unknown) model behaviour and model misbehaviour. Search-Based Software Testing (SBST) [23] and reverse engineering tools are some suggested approaches that might be adopted to search for agent-

based models' parameters that can detect (kill) mutants and learn model behaviour which can be reinforced by expert decision making and human interaction.

The combinatorial explosion that might be caused by the use of SBST and the most probably enormous number of agent interactions might be addressed by using a divide and conquer strategy that aims to test smaller portions of an ABM or substituting the agent-based model with a simpler prototype or metamodel. This last approach will most likely entail a loss of precision and model fidelity in most scenarios and would require more investigations.

Acknowledgments

This research is supported by EPSRC grant EP/G009600/1 (Automated Discovery of Emergent Misbehaviour).

References

[1] 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. Retraction of Reyes and Chang, Science 308 (5724) 1028-1031. Retraction of Chang and Roth, Science 293 (5536) 1793-1800," *Science*, vol. 314, p. 1875, 2006.

[2] L. Hatton, "The t experiments: errors in scientific software," *IEEE Computational Science and Engineering*, vol. 4, pp. 27–38, 1997.

[3] K. Simons, "Model error - evaluation of various finance models," *New England Economic Review*, pp. 17–28, 1997.

[4] 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*, vol. 4, pp. 1077–1092, 2007.

[5] J. H. Holland and J. H. Miller, "Artificial adaptive agents in economic theory," *The American Economic Review*, vol. 81, no. 2, pp. 365–370, 1991.

[6] K. Bentley, H. Gerhardt, and P. Bates, "Agent-based simulation of notch-mediated tip cell selection in angiogenic sprout initialisation," *Journal of Theoretical Biology*, vol. 250, pp. 25–36, 2008.

[7] W. M. Bulleit and M. W. Drewek, "An agent-based model of terrorist activity," in *Proceedings of the North American Association for Computational Social and Organizational Science (NAACSOS 2005)*, 2005.

[8] K. Kollman, J. H. Miller, and S. E. Page, "Adaptive parties in spatial elections," *The American Political Science Review*, vol. 86, no. 4, pp. 929–937, 1992.

[9] 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.

[10] "FLAME: Flexible Large-scale Agent-based Modelling Environment, http://www.flame.ac.uk."

[11] M. Wooldridge and N. Jennings, "Intelligent agents: Theory and practice," *Knowledge Engineering Review*, vol. 10, no. 2, pp. 115–152, 1995.

[12] M. Wooldridge, *An Introduction to MultiAgent Systems*. Wiley, 2002.

[13] J. Kidney and J. Denzinger, "Testing the limits of emergent behavior in mas using learning of cooperative behavior," in *Proc. of the 17th European Conference on Artificial Intelligence (ECAI)*, 2006, pp. 260–264.

[14] J. Denzinger and J. Kidney, "Evaluating different genetic operators in the testing for unwanted emmergent behavior using evolutionary learning of behavior," in *Proc. of the International Conference on Intelligent Agent Technology (IAT)*, 2006, pp. 23–29.

[15] N. R. Jennings, K. Sycara, and M. Wooldridge, "A roadmap of agent research and development," *Journal of Autonomous Agents and Multi-Agent Systems*, vol. 1, no. 1, pp. 7–38, 1998.

[16] J.-M. Galán, L.-R. Izquierdo, S.-S. Izquierdo, J.-I. Santos, R. del Olmo, A. López-Paredes, and B. Edmonds, "Errors and artefacts in agent-based modelling," *Journal of Artificial Societies and Social Simulation (JASSS)*, vol. 12, no. 1, 2009.

[17] S. Kim, J. A. Clark, and J. A. McDermid, "Class mutation: Mutation testing for object-oriented programs," in *Proceedings of the Net.ObjectDays Conference on Object-Oriented Software Systems*, 2000.

[18] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "Mujava : An automated class mutation system," *Journal of Software Testing, Verification and Reliability*, vol. 1, no. 2, pp. 97–133, 2005.

[19] D. Schuler and A. Zeller, "Javalanche: efficient mutation testing for java," in *Proc of the 7th joint meeting of the European Software Engineering Conference (ESEC)and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2009, pp. 297–298.

[20] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, "Interface mutation: An approach for integration testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 3, pp. 228–247, 2001.

[21] E. Merelli and M. Young, "Validating MAS models with mutation," in *First International Workshop on Multi-agent systems for Medicine, Computational biology and Bioinformatics. AAMAS*, 2005.

[22] L. Shan and H. Zhu, "Testing software modelling tools using data mutation," in *International workshop on Automation of software test*, 2006, pp. 43–49.

[23] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.