WES: Agent-based User Interaction Simulation on Real Infrastructure

John Ahlgren, Maria Eugenia Berezin, Kinga Bojarczuk, Elena Dulskyte, Inna Dvortsova,
Johann George, Natalija Gucevska, Mark Harman, Ralf Lämmel, Erik Meijer, Silvia Sapora,
Justin Spahr-Summers*
FACEBOOK Inc.

ABSTRACT

We introduce the Web-Enabled Simulation (WES) research agenda, and describe FACEBOOK's WW system. We describe the application of WW to reliability, integrity and privacy at FACEBOOK[1], where it is used to simulate social media interactions on an infrastructure consisting of hundreds of millions of lines of code. The WES agenda draws on research from many areas of study, including Search Based Software Engineering, Machine Learning, Programming Languages, Multi Agent Systems, Graph Theory, Game AI, and AI Assisted Game Play. We conclude with a set of open problems and research challenges to motivate wider investigation.

1 INTRODUCTION

A Web-Enabled Simulation (WES) is a simulation of the behaviour of a community of users on a software platform. It uses a (typically web-enabled) software platform to simulate real-user interactions and social behaviour on the real platform infrastructure, isolated from production users. Unlike traditional simulation [32, 41], in which a model of reality is created, a WES system is thus built on a real–world software platform.

In order to model users' behaviour on a WES system, a multi agent-based approach is used, in which each agent is essentially a bot that simulates user behaviour. This user behaviour could be captured in a rule-based system, or could be learnt, either supervised from examples, or unsupervised in a reinforcement learning setting.

The development of approaches to tackle the challenges posed by WES may thus draw heavily on recent advances in machine learning for software games, a topic that has witnessed many recent breakthroughs [51].

---

*Author order is alphabetical. Correspondence to Mark Harman (markharman@fb.com).
[1]'FACEBOOK', refers to the company, while 'Facebook' refers to the specific product.

In this paper we set out the general principles for WES systems. We outline the development of FACEBOOK's WW simulation of social media user communities, to illustrate principles of and challenges for the WES research agenda. WW is essentially a scaled down simulation of FACEBOOK's platform, the actions and events of which use the same infrastructure code as the real platform itself. We also introduce two new approaches to testing and optimising systems: Social Testing and Automated Mechanism Design.

Social Testing tests users' interactions with each other through a platform, while Automated Mechanism Design combines Search Based Software Engineering (SBSE) and Mechanism Design to automatically find improvements to the platforms it simulates.

Like any software testing system, the WES approach helps find and fix any issues, e.g., with software changes and updates. In common with testing systems more generally, WES operates in a safely isolated environment. The primary way in which WES builds on existing testing approaches lies in the way it models behaviour. Traditional testing focuses on system behaviour rather than user behaviour, whereas WES focuses on the interactions between users mediated by the system.

It is a subtle shift of emphasis that raises many technical and research challenges. Software systems involve increasing levels of social interaction, thereby elevating the potential for issues and bugs relating to complex interactions between users and software. It is the emergence of these kinds of social bugs and issues that necessitate the need for a WES-style approach, and the research agenda that underpins it. FACEBOOK's WW simulation is WES that uses bots that try to break the community standards in a safe isolated environment in order to test and harden the infrastructure that prevents real bad actors from contravening community standards.

Widespread Applicability: Community behaviour is increasingly prevalent in software applications, for example for travel, accommodation, entertainment, and shopping. These systems use social interactions so that each user can benefit from the collective experience of other users. Although this paper focuses on FACEBOOK's WW system, the concepts and approach could also find application in platforms used by other organisations.

Realism: WES interactions between bots are achieved through the *real* platform infrastructure, whereas a more traditional simulation approach would first *model* this infrastructure. This is important because the platform infrastructures that mediate user interactions are increasingly complex. For instance, FACEBOOK's WW simulation is built on a social media infrastructure consisting of several hundreds of millions of lines of code. While a traditional simulation modelling approach is applicable, there are many issues that are better understood using a WES approach, as we shall see.

Platform realism does not necessarily mean that the interactions between users need to be realistic representations of the end users' experience. A WES system could, for instance, allow engineers to experiment with new features for which, by definition, there is no known realistic user behaviour. It can also be used to focus on atypical behaviours of special interest to engineers, such as those of bad actors. A WES system could even be used for counterfactual simulation; modelling what users *cannot* do. We use the terms 'platform realism' and 'end user realism' for clarity. The term 'platform realism' refers to the degree to which the simulation uses the real platform. The term 'end user realism' refers to the degree to which simulated interactions faithfully mimic real users' interactions. The former is inherent to the WES approach, while the latter may be desirable, but is not always essential.

Opportunities for Researchers: It may not be possible for researchers to experiment with WES systems directly (for example, where they are built from proprietary software platforms). Nevertheless, many open questions can be tackled using traditional simulation, with results extended or extrapolated for application to WES systems. Researchers can also experiment with and evaluate novel techniques and approaches using WES systems built from open source components.

We report on our plans for the further future development of WW. The WES research agenda resides at an exciting intersection between many topics including, but not limited to, Search Based Software Engineering (SBSE) [24], Multi Agent Systems [54], Machine Learning [47], Software Testing [7], Graph Theory [53], Game Theory [42], and Game AI [55]. We hope that this paper will serve to stimulate interest in activity in the development of research on WES approaches.

The primary contributions of this paper are:

(1) The introduction of the WES approach to on-platform simulation of real-world software user communities;

(2) The introduction of the concepts of Automated Mechanism Design and Social Testing, both of which are relevant to WES systems, but also have wider applications;

(3) An outline of the FACEBOOK WW system; an example of a WES system, applied to social media user communities;

(4) A list of open problems and challenges for the WES research agenda.

2 WEB-ENABLED SIMULATION

A WES simulation can be seen as a game, in which we have a set of players that operate to fulfil a certain objective on a software platform. This observation connects research on WES systems with research on AI Assisted Game Play [51]. In an AI Assisted Game, reinforcement learning can be used to train a bot to play the game, guided by a reward (such as winning the game or achieving a higher score). Similarly, a WES simulation can also use reinforcement learning to train bots.

For example, with FACEBOOK's WW simulation, we train bots to behave like bad actors, guided by rewards that simulate their ability to behave in ways that, in the simulation, contravene our community standards [15]. The users whose behaviour is stimulated by other WES approaches could be end-users of the software platform but, more generally, could also be any software user community.

For example, a simulation of the users of a continuous integration system, would be a simulation of a developer community, while an App Store WES system may involve both developers and end users. We define the following generic concepts that we believe will be common to many, if not all, WES systems:

Bot: A bot is an autonomous agent. Note that bots can create 'new' data as they execute. For instance, social networking bots may weave connections in a social graph as they interact, 'just as the Jacquard loom weaves flowers and leaves' [34].

Action: An action is a potentially state-changing operation that a bot can perform on the platform.

Event: An event is a platform state change that is visible to some set of users.

Observation: An observation of the platform's state does not change the platform state. It is useful to distinguish actions (state changing), from observations (which are pure functional). This means that some apparent observations need to be decomposed into action and observation pairs. For example, the apparent observation 'read a message', may update notifications (that the message has been read). Therefore, it is decomposed into the (state-changing) action of getting the message (which happens once) and the observation of reading the message (which is side–effect free and can occur multiple times for those messages that permit multiple reads).

Read-only bot: a read-only bot is one that cannot perform any actions, but can observe state. Read-only bots can potentially operate on real platform data, because they are side–effect free, by construction.

Writer bot: a writer bot that can perform actions and, thereby, may affect the platform state on which it acts (e.g. the social graph in the case of social media applications).

Fully isolated bot: a fully isolated bot can neither read from not write to any part of state that would affect real user experience, by construction of the isolation system in place (See Section 2.2).

Mechanism: The mechanism is the abstraction layer through which a bot interacts with the platform. Actions, events and observations may be restricted and/or augmented by the mechanism. For instance, the mechanism might constrain the actions and events so that the bot can only exhibit a subset of real behaviours of particular interest, such as rate limiting, or restricted observability. The mechanism might also augment the available actions and events to explore novel approaches to the platform configuration, products and features, before these are released to end users. This opens up the potential for automated mechanism design, as we discuss in Section 2.3. The mechanism is also one way in which we achieve isolation (see Section 2.2).

One obvious choice for a mechanism would be to provide a bot with a user interface similar to the one that the GUI offers to a real user. However, there is no need for a WES system to interface through the GUI. There may be practical reasons why it is advantageous to implement interactions more deeply within the infrastructure. For example, it is likely to be more efficient for a WES bot to bypass some levels of abstraction between a normal user and the core underlying platform. Of course, there is a natural tension between the twin objectives of platform realism and efficiency, a topic to which we return in Section 5.

Script: A script run is akin to a single game episode. Scripts capture, *inter alia*, the type of environment created, how bots interact, simulation stopping criteria and measurement choices.

Simulation time: The simulation may compress or expand the apparent time within the simulation in order to simulate a desired use case more efficiently or effectively. However, there will always be a fundamental limitation, because the simulation can only be as fast as the underlying real platform environment will permit. This is another difference between WES and traditional simulation.

Monitor: The monitor captures and records salient aspects of the simulation for subsequent analysis.

2.1 Bot Training

Bots behave autonomously, though training to exhibit particular behaviours of interest. In the simplest use case, bots merely explore the platform, randomly choosing from a predefined set of actions and observations. More intelligent bots use algorithmic decision making and/or ML models if behaviour. The bots could also be modelled to cooperate towards a common goal.

2.2 Bot Isolation

Bots must be suitably isolated from real users to ensure that the simulation, although executed on real platform code, does not lead to unexpected interactions between bots and real users. This isolation could be achieved by a 'sandbox' copy of the platform, or by constraints, e.g., expressed through the mechanism and/or using the platform's own privacy constraint mechanisms.

Despite this isolation, in some applications bots will need to exhibit high end user realism, which poses challenges for the machine learning approaches used to train them. In other applications where read only bots are applicable, isolation need not necessarily prevent the bots from reading user information and reacting to it, but these read only bots cannot take actions (so cannot affect real users).

Bots also need to be isolated from affecting production monitoring and metrics. To achieve this aspect of isolation, the underlying platform requires (some limited) awareness of the distinction between bots and real users, so that it can distinguish real user logging data from that accrued by bot execution. At FACEBOOK, we have well-established gate keepers and configurators that allow us to do this with minimal intervention on production code. These gate keepers and configurators essentially denote a Software Product Line [12] that could also, itself, be the subject of optimisation [22].

Finally, isolation also requires protection of the reliability of the underlying platform. We need to reduce the risk that bots' execution could crash the production system or affect it by the large scale consumption of computational resources.

2.3 Automated Mechanism Design

Suppose we want to experiment with the likely end user behavioural response to new privacy restrictions, before implementing the restrictions on the real platform. We could fork the platform and experiment with the forked version. However, WES offers a more 'light touch' approach: we simply adjust the *mechanism* through which bots interact with the underlying platform in order to model the proposed restrictions. The mechanism can thus model a possible future version of the platform.
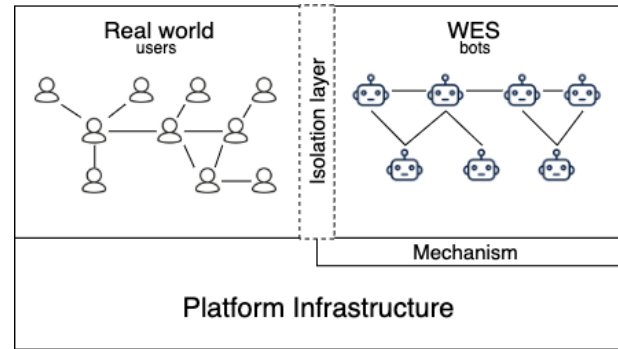

Figure 1: Generic WES System Infrastructure. Real users and bots reside on the same overall platform infrastructure. There is a conceptual isolation layer between them that determines the level of interaction possible, if any, between bots and real users. There is also a mechanism layer that mediates the actions and observations that the bots can perform through the platform.

Like all models, the mechanism need not capture all implementation details, thereby offering the engineer an agile way to explore such future platform versions. The engineer can now perform A/B tests through different mechanism parameters, exploring the differential behaviours of bot communities under different changes.

Using this intermediate 'mechanism' layer ameliorates two challenges for automated software improvement at industrial scales: build times and developer acceptance. That is, build times can run to minutes [6] or even hours [27], so an automated search that insists on changes to the code may incur a heavy computational cost. Even where cost is manageable, developers may be reluctant to land code changes developed by a machine [45]. We have found that a 'recommender system' approach sits well with our developers' expectations at FACEBOOK [38]. It ensures that the developer retains ultimate control over what lands onto the code base.

The ease with which the mechanism can be adjusted without needing to land changes into the underlying platform code means that this exploration process can be automated. Automated Mechanism Design is thus the search for optimal (or near optimal) mechanisms, according to some fitness criteria of interest. In the domain of AI Assisted Game Play, this is akin to changing the rules of the game as the player plays it, for example to make the game more challenging for an experienced player [33].

Borrowing the terminology of economic game theory [28], we use the term 'Automated Mechanism Design' to characterise the automated (or semi automated) exploration of the search space of possible mechanisms through which WES bots interact with the underlying infrastructure. Automated Mechanism Design is therefore also another application of Search Based Software Engineering (SBSE) [23, 24]. As with AI Assisted Game Play, we wish to make the platform more challenging, for example, to frustrate bad actors. However, the applications of Automated Mechanism Design are far wider than this because it offers a new approach to automated A/B testing, at volumes never previously considered.

2.4 Social Testing

WES systems bear some relationships to testing, in particular, end-to-end system level testing. Indeed, FACEBOOK's WW traces its origins back to observations of the behaviour of multiple executions of FACEBOOK's Sapienz automated test design platform [3]. However, even with only a single bot, a WES system differs from traditional testing, because a WES bot is trained, while a traditional test follows a specific sequence of input steps.

Furthermore, unlike end-to-end tests, which typically consider the journey of a *single* user through the system and avoid test user interaction lest it elevate test flakiness [25], a WES system specifically *encourages* test user interaction to model community behaviour. Therefore, WES systems can reveal faults that are best explored and debugged at this 'community' level of abstraction. We give several examples of such faults, encountered in our work at FACEBOOK. Our analysis of the most impactful production bugs indicated that as much as 25% were social bugs, of which at least 10% has suitable automated oracles to support a WES approach.

Such social bugs, arising through community interactions, require a new approach to testing: *Social Testing*; testing emergent properties of a system that manifest when bots interact with one another to simulate real user interactions on the platform. WES systems are clearly well-suited to Social Testing. However, we believe other approaches are also viable; Social Testing is an interesting new level of abstraction (lying above system testing levels of abstraction). It is worthy of research investigation in its own right.

In theory, all such 'social faults' could be found at the system level. Indeed, all *system* level faults could, *in theory*, be found at *unit* level. In practice, however, it proves necessary to stratify testing. We believe that social testing (at the whole platform level) is just a new level of abstraction; one that is increasingly important as systems themselves become more social.

3 FACEBOOK'S WW

At FACEBOOK, we are building a WES system (called WW), according to the principles set out in Section 2. WW is an environment and framework for simulating social network community behaviours, with which we investigate emergent social properties of FACEBOOK's platforms. We are using WW to (semi) automatically explore and find new improvements to strengthen Reliability, Integrity and Privacy on FACEBOOK platforms. WW is a WES system that uses techniques from Reinforcement Learning [49] to train bots (Multi Agent Reinforcement Learning) and Search Based Software Engineering [24] to search the product design space for mechanism optimisations: Mechanism Design.

Bots are represented by test users that perform different actions on real FACEBOOK infrastructures. In our current implementation, actions execute only the back-end code: bots do not generate HTTP requests, nor do they interact with the GUI of any platform surface; we use direct calls to internal FACEBOOK product libraries. These users are isolated from production using privacy constraints and a well-defined mechanism of actions and observations through which the bots access the underlying platform code. However, when one WW bot interacts with another (e.g., by sending a friend request or message) it uses the production back-end code stack, components and systems to do so, thereby ensuring platform realism.

3.1 Training WW bots

To train bots to simulate real user behaviour, we use Reinforcement Learning (RL) techniques [49]. Our bot training top level approach is depicted in Figure 2. As can be seen from Figure 2 the WW bot training closely models that of a typical RL system [49]. That is, a bot executes an action in the environment, which in turn, returns an observation (or current state), and an eventual reward to the bot. Using this information, the bot decides to take an action, and thus the SARSA (State-Action-Reward-State-Action) loop is executed during a simulation.

However, when considering the environment, we explicitly tease apart the mechanism from the underlying platform. The platform is out of WW's control: its code can change, since it is under continual development by developers, but WW cannot change the platform code itself. Furthermore, WW cannot determine the behaviour of the platform. The platform may choose to terminate and/or it may choose to allocate different resources on each episode of the simulation. Furthermore, the social graph at the heart of the database is also continually changing.

The mechanism helps to maintain a consistent interface for WW bots, so that their code is insulated from such changes. It also mediates the actions and observations a bot can make and witness, so that many different mechanisms can be explored without any need to change the platform. As can be seen from Figure 2, the mechanism is separated from the platform. Each bot contains its own set of mechanism parameters, so that each can report the fitness of a different mechanism parameter choice. At the same time, the bots seek to achieve their goals, guided by reinforcement learning.

For example, to simulate scammers and their targets, we need at least two bots, one to simulate the scammer and another to simulate the potential target of the scam. The precise technical details of how we impede scammers on WW are obviously sensitive, so we cannot reveal them here. Nevertheless, we are able to outline the technical principles.

The reinforcement learning reward for the scammer bot is based on its ability to find suitable candidate targets, while the candidate targets are simulated by rule-based bots that exhibit likely target behaviours. The mechanism parameters residing in the scammer bots are used to compute fitness in terms of the mechanism's ability to impede the scammers in their goal of finding suitable targets.

This use case need not involve a large number of bots. However, the ability to simulate at scale gives us automated parallelisation for increased scalability of the underlying search, and also supports averaging fitness and reward results over multiple parallel executions. Such averaging can be useful for statistical testing since results are inherently stochastic.

3.2 Top Level Implementation

The top level components of the FACEBOOK WW system are depicted in Figure 3. This is a very high level view; there is, of course, a lot more detail not shown here. We focus on key components to illustrate overall principles. WW consists of two overall subsystems: the general framework classes, which are the core of our simulation platform (and remain unchanged for each use case), and the per-use-case classes (that are tailored for each use case).
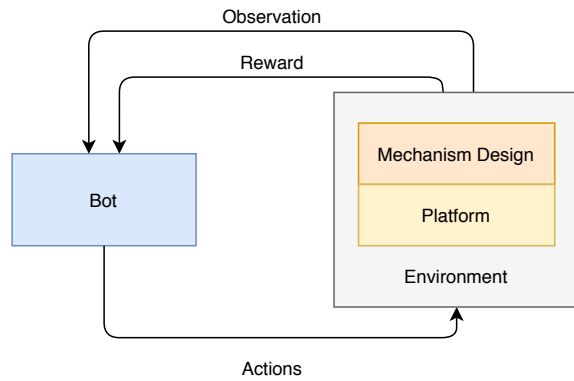
Figure 2: WW Reinforcement Learning Architecture: bots execute actions in the environment, which in turn, returns an observation, and an eventual reward. The platform's code is unchanged by simulation, while the mechanism through which the bots interact with the platform is subject to change during the simulation process (to explore optimisations of the underlying platform).
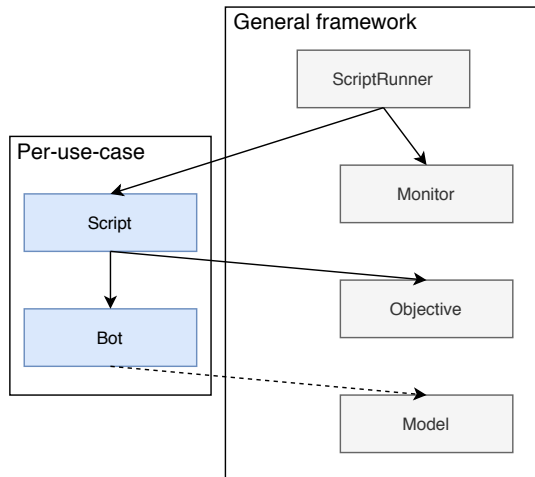

Figure 3: WW implementation: general framework components form the core platform, while per-use-case components are specialised to each use case.

3.2.1 General Framework classes.

ScriptRunner: entry point to the WW simulation. It is responsible for building the environment necessary for a WW script, executing the state machine, and setting up monitoring of the results.

Monitor: responsible for recording events and collecting data for post-analysis, as the Script is run.

Objective: represents an objective that a Script is aiming to achieve. Possible objectives include time, steps, episodes, 'results' (such as vulnerabilities found, etc.). The objective is also used to determine when to end the simulation.

Algorithm 1: Pseudo-code of the ScriptRunner loop

1	**Setup**
2	Script creates the environment and the Bots.
3	**while** *Objective is not reached* **do**
4	Advance the virtual-time clock.
5	Execute the action of the next Bot.
6	Observe and log events and data.
7	**End**
8	Finalize the simulation (cleanup).

Model: a machine learning model for the bot, e.g., a Policy Gradient model with determined parameters.

3.2.2 Per-use-case classes.
The core WW platform consists of the general framework class together with a set of components from which the per-use case classes are defined. In order to define each use case, we simply define a script and a bot class, using the components and deploy them on the general framework.

Script: describes the user community (e.g., the size of the graph), and the environment where the users will interact (e.g., groups with fake news).

Bot: an automated agent (represented by a test user) with a particular behaviour defined by actions. For example, a FACEBOOK Messenger user. A bot interacts with other users (as defined by its behaviour), and can have its own learning model.

4 APPLICATIONS OF WW AT FACEBOOK

We believe many of these application use cases for WW may generalise to other WES systems, but we give them here in their FACEBOOK context to illustrate WES applications with specific concrete use cases. At the time of writing we are focusing our engineering effort on the applications of WW to integrity challenges, but we fully anticipate application to the other areas listed in the section in due course. Indeed, we expect many more use cases to emerge as the development of the WW infrastructure matures.

4.1 Integrity and Privacy

In any large scale system, not all user behaviour will be benign; some behaviours are perpetrated by bad actors, intent on exploiting the platform and/or its users. On the Facebook platform such bad actor user behaviour includes any actions that contravene FACEBOOK's Community Standards [15].

We are using WW to enhance our ability to detect bad actor behaviours. We are also developing Automated Mechanism Design approaches that search product design space to find ways to harden the platform against such bad actors, thereby promoting the integrity of the platform and its users. In this section, we illustrate both with applications of WW to the challenges of detecting and preventing contravention of integrity constraints.

WW also provides us with a realistic, yet safely isolated, way to investigate potential privacy issues on the real platform. Because the WW bots are isolated from affecting real users, they can be trained to perform potentially privacy–violating actions on each other.

On the other hand, because the bots use the real infrastructure, the actionability of any potential issues found through such simulation is considerably increased (compared to a traditional simulation on a purely theoretical model).

Simulating bad actors: Consider the problem of users trying to post violating content on the Facebook platform. Even though we employ state-of-the-art classifiers and algorithms to protect the platform, we need to be proactive in our exploration of the space of potential exploits; WW provides one way to do this. If our bots succeed in finding novel undetected contravening behaviours, the WW simulation has allowed us to head off a potential integrity vulnerability.

Search for bad content: Bad actors use our platform to try to find policy–violating content, or to try to locate and communicate with users who may share their bad intent. Our bots can be used to simulate such bad actors, exploring the social graph. This enables them, for example, to seek out policy–violating content and the violators that create it. WW bots can also search for clusters of users sharing policy–violating content.

Searching for mechanisms that impede bad actors: We use automated mechanism design to search for ways to frustrate these bad actors from achieving their goals within the simulation. This makes the optimisation problem a multi–objective one, in which the bots seek to achieve bad actions, while the system simultaneously searches for mechanisms that frustrate their ability to do so. The search is also multi objective because we require new mechanisms that simultaneously frustrate the achievement of bad actions, while having little or no noticeable impact on normal users. In this way we use WW automated mechanism design to explore the space of potential changes that may also lead to greater integrity of the real platform.

Interestingly, this is a problem where *preventing* bad activity does not require the ability to *detect* it. Automated search may yield mechanisms that frustrate scamming, for example by hiding potential victims from scammers, without necessarily relying on classifiers that detect such scammers. This is reminiscent of the way in which *removing* side effects (which may be computable), does not require the ability to *detect* side effects (which is undecidable, in general) [21].

Bots that break privacy rules: In the Facebook infrastructure, every entity has well–defined privacy rules. Creating bots trained to seek to achieve the sole purpose of breaking these privacy rules (e.g., to access another bot's private photos) is thus a way to surface potential bugs, as well as unexpected changes in the system's behaviour. For example, if a bot was never previously able to perform a certain action (e.g., access another bot's message), but becomes able to do so after a code change, this could highlight a change in privacy rules that resulted in unexpected behaviours.

Data acquiring bots: Even with the privacy rules currently in place, a Facebook user has the ability to access another users' data (with consent, of course). This functionality is a necessary part of the normal usability of the platform. Nevertheless, we need to maintain a constantly vigilant posture against any potential to exploit this fundamentally important ability. By creating bots whose sole purpose is to accrue as much data as possible from each other, we are able to test our preventative measures and their effectiveness against this type of behaviour.
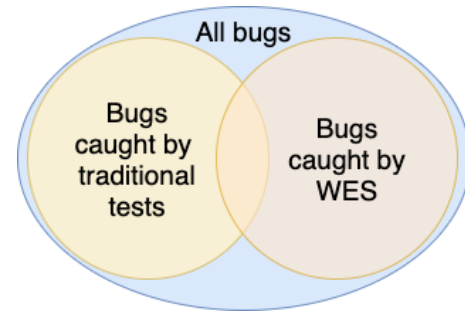

Figure 4: Traditional testing methods such as unit tests, or end-to-end tests, only succeed in capturing a subset of all possible bugs. WW **gives developers the option to test on a new level of abstraction, one that takes community behaviour and interactions into account.**

4.2 Reliability

Large organisation like Facebook naturally face challenges of reliability at scale. WW is not only a simulation of hundreds of millions of lines of code; it is a software system that runs on top of those very same lines of code. In order to cater for the reliability of the WW system itself, we use a DevOps approach, commonly used throughout the company [16]. WW runs in continuous deployment as a production version, underpinned by suitable maintenance procedures, such as time series monitoring and analysis, alarms and warnings and on-call rotations. However, WW can also be used to explore the reliability of the platform as we outline in this section.

Social Bugs: WW provides tools for social testing, whereby failures can be expressed as *percentages*, rather than more traditional binary success/fail. All traditional tests might execute successfully, yet we still observe an 'social bug' issue in production. Examples include drops in top line product metrics, significant changes in machine learning classification outcomes, big jumps in data pipeline line breakages.

These kind of bugs have many causes including code, data and/or configuration changes. While all could, *in theory* be detected by lower levels of test abstraction, it is useful to have a WES style final 'full ecosystem' test (as opposed to 'full system' test). With WW we can detect such a significant metric change before such a change affects real users, because it tests the whole ecosystem with a simulation of the community that uses the platform. A *single user* test, even at full system level, would be insufficiently expressive to capture community interaction faults.

We also retain lower levels of testing abstraction. The WW simulation is the most computationally expensive form of testing we have, as well as the highest level of abstraction. Also, although 'platform realism' is the goal of all WES systems, there are necessary compromises to achieve scalability, as discussed in Section 5.

The WES Test Oracle: These metrics play the role of test oracle [5], thereby ensuring that the platform level testing problem can be entirely automated. Of course, since WW is a scaled down version of the real community, there is a need to tune such metrics and alerts, but this is, itself, an interesting machine learning problem.

5 OPEN PROBLEMS AND SOME RELATED WORK THAT MAY HELP TACKLE THEM

In this section we review related work and highlight open problems and challenges for the WES research agenda. Neither our characterisation of related work, nor our list of open problems is comprehensive. We are excited to work with the academic and scientific research community to tackle these open problems together using such related work and/or other promising approaches.

Naturally, we can expect research to draw on the extensive body of work on simulation, and in particular, multi agent simulation [41], which is now relatively mature, with the advent of so-called 'sophisbots' that are hard to distinguish from real users [8].

There are also frameworks for simulation of software systems and communities, but these tend to focus on traditional simulation rather than on-platform simulation, the *sine qua non* of a WES system. For example, RecSym [29] uses an abstraction of a generic Recommender System (RS) infrastructure to simulate the behaviour of different choices of Reinforcement Learning (RL) for recommending content to users. The most important difference, is that WW uses RL (and other techniques) to train bots to behave like users so that the behaviours of users on the real Facebook infrastructure can be better simulated, whereas RecSym simulates the behaviour of an abstraction on Infra with respect to a given RL.

5.1 Open Problems and Challenges

Since WES systems, more generally, rely on training bots to simulate real users on real software platforms, there is a pressing need for further research on a number of related topics. This section lists 15 areas of related work that can contribute to tackling open WES research agenda challenges. The large number and diversity of topics and challenges underscores the rich opportunities for research.

Another Application for MARL: Recent developments in Multi Agent Reinforcement Learning (MARL) [30] may offer techniques that can be adapted and applied to WES systems. One important challenge is to find ways to train bots to behave like specific classes of bad actors.

Multi Objective Search: Typically, Software Engineering problems, such as reliability and integrity, will have a multi objective character. For example, it would be insufficient to constrain a mechanism to frustrate bad actors, if one does not counter-balance this objective against the (potentially competing) objective of not impeding normal users in their routine use of the platform. Fortunately, multi objective optimisation algorithms, such as NSGA II [13], are readily available and have been widely–studied in the Software Engineering community for over two decades [23]. More research is needed on the application of multi objective search to WES problems.

AI Assisted Game Play: The WES agenda naturally draws on previous work on artificial intelligence for software game play. Recent advances on competitive game playing behaviour [51] may be adapted to also imbue WES bots with more realistic social interactions. In a WES system we do not necessarily need competitive 'play', but realistic social interaction; the rewards and fitness functions may differ, but key insights may, nevertheless, carry over between these two related application paradigms.

In some WES applications it may also be important to avoid the bots acquiring super-human abilities, such as interacting faster than any human ever could. This is also a problem recently tackled in machine learning for computer game playing optimisation [51].

Automated Mechanism Design: Mechanism Design is a form of automated game design, a topic that has been studied for over a decade [50], and that remains an active area of research [33]. The challenge is to define techniques for deployment, testing and for efficiently and effectively searching the mechanism space. Tackling these problems may draw on previous work on Genetic Improvement [45], Program Synthesis [18], Constraint Solving [31], and Model Checking [11].

Co-evolutionary Mechanism Learning: Automatically improving the platform mechanism to frustrate some well-known attack from a class of bad actors may yield short term relief from such bad actors. However, in order to continue to remain 'ahead of the game' and to thereby frustrate all possible, as yet unknown attacks, we need a form of co-evolutionary optimisation; the mechanism is optimised to frustrate bad actions, while the bots simultaneously learn counter strategies that allow them to achieve these bad actions *despite* the improved mechanism.

Co-evolutionary optimisation is well-suited to this form of 'arms race'. Co-evolutionary optimisation has not yet received widespread attention from the SBSE community, although there is some initial literature on the topic [1, 4, 46]. Co-evolutionary Mechanism Design therefore establishes an new avenue of research that promises to widen the appeal and application of co-evolutionary approaches to software engineering problems.

End User Realism and Isolation: In some applications, WES bots will need to be trained to faithfully model the behaviours of the platform's real users; 'end user realism'. Tackling this may rely on recent advances in machine learning, but will also be constrained by the need for user privacy. There is also interesting research to be done on the degree of end user realism required, and metrics for measuring such realism, a problem only hitherto lightly explored in testing research [2, 9, 14, 37].

Because bots are isolated from real users, we face the research challenge of defining, capturing, measuring, and replicating realistic behaviour. Faithfully replicating every aspect of end user behaviour is seldom necessary. Indeed, in some cases, end user realism is not required at all. For example, for social testing the FACEBOOK Messenger system, we found that it was sufficient to have a community of bots regularly sending messages to one another in order to detect some social bugs that manifest through drops in production metrics, such as number of messages sent.

For integrity-facing applications, such as preventing bad actors' harmful interaction with normal users, we need reasonably faithful bad actor bot behaviours, and bots that faithfully replicate normal users' responses to such bad actors. This is a challenging, but highly impactful research area.

Search Based Software Engineering: Many of the applications of WES approaches lie within the remit of software engineering, and it can be expected that software engineering, and in particular, Search Based Software Engineering (SBSE) [23] may also find application to open problems and challenges for WES systems. In common with SBSE, WES systems share the important salient characteristic that the simulation is executed on the real system itself.

It is therefore 'direct'. This directness is one advantage of SBSE over other engineering applications of computational search [20]. We can expect similar advantages for WES systems. By contrast, traditional simulation tends to be highly indirect: The simulation is not built from the real system's components, but as an abstraction of a theoretical model of the real system and its environment.

Diff Batching: The WES approach has the advantage that it allows engineers to investigate properties of proposed changes to the platform. However, for larger organisations, the volume of changes poses challenges itself. For example, at FACEBOOK, over 100,000 modifications to the repository land in master every week [3]. Faced with this volume of changes, many companies, not just FACEBOOK [43], use Diff batching, with which collections of code changes are clustered together. More work is needed on smarter clustering techniques that group code modifications (aka Diffs) in ways that promote subsequent bisection [43].

Speed up: Simulated clock time is a property under experimental control. It can be artificially sped up, thereby yielding results in orders of magnitude less real time than a production A/B test [48]. However, since a WES system uses real infrastructure, we cannot always speed up behaviour without introducing non-determinism: If bots interact faster (with the system and each other) this may introduce race conditions and other behaviours that would tend to be thought of as flakiness in the testing paradigm [25, 35].

Social Testing: Section 2.4 introduces a new form of software testing, which we call 'Social Testing'. Testing is generally regarded as an activity that takes place at different levels of abstraction, with unit testing typically being regarded as lowest level, while system level testing is regarded as highest level. Social testing adds a new level of abstraction above system level testing. There are so many interesting problems in social testing that a complete treatment would require a full paper in its own right. In this brief paper we hope we have sufficiently motivated the introduction of this new higher level of abstraction, and that others will be encouraged to take up research on social testing.

Predictive Systems: WES systems would benefit from automated prediction (based on the simulation) of the future properties of the real world. This will help translate insights from the simulation to actionable implications for the real world phenomena. Therefore, research on predictive modelling [10, 19] is also highly relevant to the WES research agenda.

Causality: To be actionable, changes proposed will also need to correlate with improvements in the real world, drawing potentially on advances in causal reasoning [44], another topic of recent interest in the software engineering community [39].

Simulating Developer Communities: Although this paper has focused on WES for social media users, a possible avenue for other WES systems lies in simulation of developer communities. This is a potential new avenue for the Mining Software Repositories (MSR) research community [26]. The challenge is to mine information that can be usefully employed to train bots to behave like developers, thereby exploring emergent developer community properties using WES approaches. This may have applications to and benefit from MSR. It is also related to topics such as App Store analysis [40], for which the community combines developers and users, and to software ecosystems [36], which combine diverse developer sub-communities.

Synthetic Graph Generation: For WW, we are concerned with the simulation of social media. Read-only bots can operate on the real social network, which is protected by isolation. However, many applications require writer bots. Naturally, we do not want WW writer bots interacting with real users in unexpected ways, so part of our isolation strategy involves large scale generation of large synthetic (but representative) graphs. This is an interesting research problem in its own right. On a synthetic graph it will also be possible to deploy fully isolated bots that can exhibit arbitrary actions and observations, without the need for extra mechanism constraints to enforce isolation.

Game Theory: A WES execution is a form of game, in which both the players and the rules of the game can be optimised (possibly in a co-evolutionary 'arms race'). Formal game theoretic investigation of simulations offers the possibility of underpinning the empirical observations with mathematical analysis. Naturally, empirical game-theoretic analysis [52] is also highly relevant. There has been recent interest in game theoretic formulations in the Software Engineering community [17]. WES systems may provide a further stimulus for this Game Theoretic Software Engineering research agenda.

6 CONCLUSION

In this paper we set out a new research agenda: Web-Enabled Simulation of user communities. This WES agenda draws on rich research strands, including machine learning and optimisation, multi agent technologies, reliability, integrity, privacy and security as well as traditional simulation, and topics in user community and emergent behaviour analysis.

The promise of WES is realistic, actionable, on-platform simulation of complex community interactions that can be used to better understand and automatically improve deployments of multi-user systems. In this short paper, we merely outline the WES research agenda and some of its open problems and research challenges. Much more remains to be done. We hope that this paper will encourage further uptake and research on this exciting WES research agenda.

7 ACKNOWLEDGEMENTS

REFERENCES

[1] Konstantinos Adamopoulos, Mark Harman, and Robert Mark Hierons. 2004. Mutation Testing Using Genetic Algorithms: A Co-evolution Approach. In *Genetic and Evolutionary Computation Conference (GECCO 2004), LNCS 3103*. Springer, Seattle, Washington, USA, 1338–1349.

[2] Sheeva Afshan, Phil McMinn, and Mark Stevenson. 2013. Evolving Readable String Test Inputs Using a Natural Language Model to Reduce Human Oracle Cost. In *International Conference on Software Testing, Verification and Validation (ICST 2013)*. 352–361.

[3] Nadia Alshahwan, Xinbo Gao, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, Taijin Tei, and Ilya Zorin. 2018. Deploying Search Based Software Engineering with Sapienz at Facebook (keynote paper). In 10th *International Symposium on Search Based Software Engineering (SSBSE 2018)*. Montpellier, France, 3–45. Springer LNCS 11036.

[4] Andrea Arcuri and Xin Yao. 2010. Co-evolutionary automatic programming for software development. *Information Sciences* (2010). https://doi.org/doi:

10.1016/j.ins.2009.12.019 To appear. Available on line from Elsevier.

[5] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (May 2015), 507–525.

[6] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. 2015. Efficient dependency detection for safe Java test acceleration. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering.* 770–781.

[7] Antonia Bertolino. 2007. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering (FOSE'07).* IEEE, 85–103.

[8] Dan Boneh, Andrew J. Grotto, Patrick McDaniel, and Nicolas Papernot. 2019. How Relevant is the Turing Test in the Age of Sophisbots? *arXiv e-prints*, Article arXiv:1909.00056 (Aug 2019), arXiv:1909.00056 pages. arXiv:cs.CY/1909.00056

[9] Mustafa Bozkurt and Mark Harman. 2011. Automatically generating realistic test input from web services. In *IEEE 6th International Symposium on Service Oriented System Engineering (SOSE 2011)*, Jerry Zeyu Gao, Xiaodong Lu, Muhammad Younas, and Hong Zhu (Eds.). IEEE, Irvine, CA, USA, 13–24.

[10] Cagatay Catal and Banu Diri. 2009. A systematic review of software fault prediction studies. *Expert systems with applications* 36, 4 (2009), 7346–7354.

[11] Edmund M Clarke Jr, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. 2018. *Model checking.* MIT press.

[12] Paul Clements and Linda Northrop. 2001. *Software Product Lines: Practices and Patterns.* Addison-Wesley Professional. 608 pages.

[13] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6 (April 2002), 182–197. Issue 2.

[14] Dirk Draheim, John Grundy, John Hosking, Christof Lutteroth, and Gerald Weber. 2006. Realistic load testing of web applications. In *Conference on Software Maintenance and Reengineering (CSMR'06).* IEEE, 11–pp.

[15] FACEBOOK, INC. 2020. Community Standards. https://www.facebook.com/communitystandards/

[16] Dror G. Feitelson, Eitan Frachtenberg, and Kent L. Beck. 2013. Development and Deployment at Facebook. *IEEE Internet Computing* 17, 4 (2013), 8–17.

[17] Carlos Gavidia-Calderon, Federica Sarro, Mark Harman, and Earl T. Barr. To appear. The Assessor's Dilemma: Improving Bug Repair via Empirical Game Theory. *IEEE Transactions on Software Engineering* (To appear).

[18] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends in Programming Languages* 4, 1-2 (2017), 1–119.

[19] Mark Harman. 2010. How SBSE Can Support Construction and Analysis of Predictive Models (Keynote Paper). In 6th *International Conference on Predictive Models in Software Engineering (PROMISE 2010).* Timisoara, Romania.

[20] Mark Harman. 2010. Search Based Software Engineering (Keynote Paper). In 13th *International Conference on Fundamental Approaches to Software Engineering (FASE 2010).* Paphos, Cyprus.

[21] Mark Harman, Lin Hu, Robert Mark Hierons, Xingyuan Zhang, Malcolm Munro, José Javier Dolado, Mari Carmen Otero, and Joachim Wegener. 2002. A Post-Placement Side-Effect Removal Algorithm. In *IEEE International Conference on Software Maintenance* (Montreal, Canada). IEEE Computer Society Press, Los Alamitos, California, USA, 2–11.

[22] Mark Harman, Yue Jia, Jens Krinke, Bill Langdon, Justyna Petke, and Yuanyuan Zhang. 2014. Search based software engineering for software product line engineering: a survey and directions for future work. In 18th *International Software Product Line Conference (SPLC 14).* Florence, Italy, 5–18.

[23] Mark Harman and Bryan F. Jones. 2001. Search Based Software Engineering. *Information and Software Technology* 43, 14 (Dec. 2001), 833–839.

[24] Mark Harman, Afshin Mansouri, and Yuanyuan Zhang. 2012. Search Based Software Engineering: Trends, Techniques and Applications. *Comput. Surveys* 45, 1 (November 2012), 11:1–11:61.

[25] Mark Harman and Peter O'Hearn. 2018. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis (keynote paper). In 18th *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2018).* Madrid, Spain, 1–23.

[26] Ahmed E Hassan. 2008. The road ahead for mining software repositories. In *2008 Frontiers of Software Maintenance.* IEEE, 48–57.

[27] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in continuous integration: assurance, security, and flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering.* 197–207.

[28] Leonid Hurwicz and Stanley Reiter. 2006. *Designing Economic Mechanisms.* Cambridge University Press.

[29] Eugene Ie, Chih-wei Hsu, Martin Mladenov, Vihan Jain, Sanmit Narvekar, Jing Wang, Rui Wu, and Craig Boutilier. 2019. RecSim: A Configurable Simulation Platform for Recommender Systems. *arXiv e-prints*, Article arXiv:1909.04847 (Sep 2019), arXiv:1909.04847 pages. arXiv:cs.LG/1909.04847

[30] Nicholas R Jennings and Michael J Wooldridge. 1996. Software agents. *IEE review* (1996), 17–20.

[31] Dongwon Kang, Jinhwan Jung, and Doo-Hwan Bae. 2011. Constraint-based Human Resource Allocation in Software Projects. *Software: Practice and Experience* 41, 5 (April 2011), 551–577.

[32] Jack PC Kleijnen. 2005. Supply chain simulation tools and techniques: a survey. *International journal of simulation and process modelling* 1, 1-2 (2005), 82–89.

[33] Kamolwan Kunanusont, Raluca D Gaina, Jialin Liu, Diego Perez-Liebana, and Simon M Lucas. 2017. The n-tuple bandit evolutionary algorithm for automatic game improvement. In *2017 IEEE Congress on Evolutionary Computation (CEC).* IEEE, 2201–2208.

[34] Ada Augusta Lovelace. 1843. Sketch of the Analytical Engine Invented by Charles Babbage By L. F. Menabrea of Turin, officer of the military engineers, with notes by the translator. (1843). Translation with notes on article in italian in Bibliothèque Universelle de Genève, October, 1842, Number 82.

[35] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In 22nd *International Symposium on Foundations of Software Engineering (FSE 2014)*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne Storey (Eds.). ACM, Hong Kong, China, 643–653.

[36] Konstantinos Manikas and Klaus Marius Hansen. 2013. Software ecosystems– A systematic literature review. *Journal of Systems and Software* 86, 5 (2013), 1294–1306.

[37] Ke Mao, Mark Harman, and Yue Jia. 2017. Robotic Testing of Mobile Apps for Truly Black-Box Automation. *IEEE Software* 34, 2 (2017), 11–16.

[38] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. 2019. SapFix: Automated End-to-End Repair at Scale. In *International Conference on Software Engineering (ICSE) Software Engineering in Practice (SEIP) track.* Montreal, Canada.

[39] William Martin, Federica Sarro, and Mark Harman. 2016. Causal Impact Analysis for App Releases in Google Play. In *24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering(FSE 2016).* Seattle, WA, USA, 435–446.

[40] William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. 2017. A Survey of App Store Analysis for Software Engineering. *IEEE Transactions on Software Engineering* 43, 9 (2017).

[41] Fabien Michel, Jacques Ferber, and Alexis Drogoul. 2018. Multi-Agent Systems and Simulation: A Survey from the Agent Commu-nity's Perspective. In *Multi-Agent Systems.* CRC Press, 17–66.

[42] Roger B Myerson. 2013. *Game theory.* Harvard university press.

[43] Armin Najafi, Peter C. Rigby, and Weiyi Shang. 2019. Bisecting Commits and Modeling Commit Risk during Testing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019).* 279–289.

[44] Judea Pearl. 2000. *Causality.* Cambridge University Press, Cambridge.

[45] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David R. White, and John R. Woodward. 2018. Genetic Improvement of Software: a Comprehensive Survey. *IEEE Transactions on Evolutionary Computation* 22, 3 (June 2018), 415–432. https://doi.org/doi:10.1109/TEVC.2017.2693219

[46] Jian Ren, Mark Harman, and Massimiliano Di Penta. 2011. Cooperative Co-evolutionary Optimization on Software Project Staff Assignments and Job Scheduling. In 3rd *International Symposium on Search based Software Engineering (SSBSE 2011).* 127–141. LNCS Volume 6956.

[47] Luis G. Serrano. [n.d.]. *Grokking Machine Learning.* Manning Publications.

[48] Dan Siroker and Pete Koomen. 2013. *A/B testing: The most powerful way to turn clicks into customers.* John Wiley & Sons.

[49] Richard S. Sutton (Ed.). 1992. *Reinforcement Learning.* SECS, Vol. 173. Kluwer Academic Publishers. Reprinted from volume 8(3–4) (1992) of Machine Learning.

[50] Julian Togelius and Jurgen Schmidhuber. 2008. An experiment in automatic game design. In *2008 IEEE Symposium On Computational Intelligence and Games.* IEEE, 111–118.

[51] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander S. Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L. Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. 2019. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature* 575, 7782 (2019), 350–354.

[52] Michael P. Wellman. 2006. Methods for Empirical Game-Theoretic Analysis. In *Proceedings of the 21st AAAI Conference.* 1552–1556.

[53] Douglas Brent West. 2001. *Introduction to graph theory.* Vol. 2. Prentice hall Upper Saddle River.

[54] Michael Wooldridge and Nicholas R Jennings. 1994. Agent theories, architectures, and languages: a survey. In *International Workshop on Agent Theories, Architectures, and Languages.* Springer, 1–39.

[55] Geogios N Yannakakis. 2012. Game AI revisited. In *Proceedings of the 9th conference on Computing Frontiers.* 285–292.