It's Good to be Different: Diversity, Heterogeneity and Dynamics in Collective Systems

Peter R. Lewis*, Harry Goldingay* and Vivek Nallur†
*School of Engineering and Applied Science, Aston University, Birmingham, UK
†School of Computer Science and Statistics, Trinity College Dublin, Ireland
{p.lewis|goldinhj}@aston.ac.uk, vivek.nallur@scss.tcd.ie

***Abstract*—We identify two different forms of diversity present in engineered collective systems, namely heterogeneity (genotypic/phenotypic diversity) and dynamics (temporal diversity). Three qualitatively different case studies are analysed, and it is shown that both forms of diversity can be beneficial in very different problem and application domains. Behavioural diversity is shown to be motivated by input diversity and this observation is used to present recommendations for designers of collective systems.**

I. Introduction

Terms such as *diversity* and *heterogeneity* have long been borrowed by computer science, from older fields such as ecology, to describe systems. In ecology, diversity usually refers to differences between species (e.g. plant diversity), while heterogeneity usually refers to variations in the environment (e.g. soil heterogeneity). However, there is significant overlap in the use of the two terms in computer science literature; both refer to the presence of variability in a system. In this paper we are concerned with systems where the designer has the freedom to decide, at least to some extent, whether and how to employ forms of diversity in an engineered collective system. Therefore, we propose a general framework for the description of how collective systems may vary, thereby proposing definitions for heterogeneity and diversity in computing systems. We then review three case studies, two from our own prior work and one novel one, in the context of this framework. We argue that forms of diversity in which differences occur *between components* and *over time* can both be beneficial in the design of computing systems, particularly with respect to the adaptability of collective systems. Our framework enables and promotes the explicit consideration of diversity, and therefore aids the engineering process.

Diversity has been advocated as a mechanism for introducing robustness in the software engineering community, for many years. Since the eighties, N-version programming [1] and design diversity [2] have been proposed to protect against faults introduced through designer assumptions and during implementation. Similar results have been found in hardware, where diversity across redundant circuits can improve fault tolerance [3]. Other researchers [4]–[6] demonstrated that diversity in various forms can be used to increase robustness in a range of different software. More recently, diversity has been used as a mechanism to guard against security holes prevalent in web-based systems [7]–[9]. Diversity has also been used as a design principle in ensemble learning, and has proved to be particularly useful in dealing with dynamic learning problems [10].

Similarly, *heterogeneity* has been described as an important characteristic of multi-agent and self-organising systems, allowing them to adapt to unknown situations or environments. Campbell et al. [11] show that introducing variation in agent behaviour to achieve dynamic task allocation increases the system's ability to adapt to varying stimuli. Prasath et al. [12] recommend heterogeneity in wireless sensor networks as a mechanism for generating near-optimal configurations. Lewis et al. [13] demonstrate that heterogeneity in distributed smart camera networks can lead to new highly efficient configurations.

Goldingay and Lewis [14] define a heterogeneous system as one in which at least two components are different at any point in time. They distinguish this from a related notion, where differences are present within one or more components over time instead of between components at a particular point in time. This *temporal diversity* has also been advocated by Lathia et al. [15] as a mechanism for retaining user interest in recommender systems. In particle swarm optimisation (PSO), Goldingay and Lewis [14] show separate performance improvements due to both heterogeneity and temporal diversity.

A common theme in these works is the deliberate introduction of differences in either behaviour or structure, to achieve an adaptive capacity that can deal with varying, possibly unknown environments or inputs.

II. Forms and Causes of Diversity

In this paper, we describe various forms of diversity that can be present in an engineered collective system, and examine the conditions under which they are beneficial. In particular, we discuss how diversity of input to a system motivates diversity within the system. Our focus is on behaviourally diverse systems, where we define behaviour to be the *observable* response by a system (or one of its components) to a set of potential inputs. Observable behaviour is not solely tied to the system's functional goals, but could also include secondary properties such as time taken, or amount of power used to complete a task. As noted in the previous section, *diversity* is a broad term and not all systems express their diversity in the same way.

Clearly the term diversity implies that a system in some sense possesses variability, but this variability can be expressed on two axes:

1) *Heterogeneity:* Differences between components of a collective system at a given point in time,

2) *Dynamics:* Differences within a single component over time.

Recent work [13], [14] has established that, while the concepts of *heterogeneity* and *dynamics* are closely related, their benefits differ. From a design perspective, it is important to clearly differentiate between the two. Inspired by distinctions made between these two forms of diversity in ecology [16], we use the term *diversity* to refer to the overarching concept of a system containing variability, and the terms *heterogeneity* and *dynamics* to clearly distinguish between the two axes on which diversity can be expressed.

Of course, differences over time (i.e. dynamics) may be replicated across many components in a collective system; many components may each independently express differences in behaviour over time. Furthermore, this may occur according to some synchronous pattern of change, such that heterogeneity is never present in the system. Alternatively, both heterogeneity and dynamics may be present simultaneously. For example, unsynchronised dynamics across a collective may indeed be a cause of heterogeneity.

In the biological world, the primary mechanism for the introduction of differences between organisms is random mutation at the genotypic level. These genotypic differences can translate to phenotypic (e.g. physiological, morphological or behavioural) differences between individuals in a population [17]. The range of genotypic and phenotypic diversity in a population therefore depends both on the rate of mutation and also on the ability of mutated individuals to compete for survival along with other individuals. In the computing literature, where the term heterogeneity has been previously used to refer to differences between components in terms of their behaviour (e.g. in [13]) or physical characteristics (e.g. in [12]), it is typically to refer to phenotypic diversity. The software engineering literature also sometimes describes these two types of diversity as *genotypic/phenotypic* and *temporal*, while the literature on collective systems often uses the terms *heterogeneity* and *dynamics* respectively.

Importantly, from an engineering point of view, having a diverse system is not a goal in itself, but is typically motivated by input diversity. That is, depending on system properties and the operating environment, differences arise between the inputs to components (*heterogeneous input*) or over time (*dynamic input*) or both. Intuitively, given given a range of possible behaviours, each suited to certain inputs but not to others, heterogeneous input motivates heterogeneous behaviour with each component effectively specialising to match its input. Similarly, dynamic input motivates a dynamic choice between behaviours. Again, from a design perspective, it is important to understand how problem and system characteristics lead to different types of input diversity and how we can best generate behavioural diversity in response.

Often, the reason for engineering systems as collectives is to introduce redundancy, in order to provide fault-tolerance. Typically, redundancy is assumed to imply the duplication of identical components, which provide identical functionality and also fail or perform poorly in the same circumstances. An alternative approach is for components to degenerate with respect to each other. Degeneracy is the ability of different components be functionally equivalent in certain cases, while being functionally different from each other in others, and has been shown to be a source of robustness [18]. The fundamental concept however remains the same, in that a difference between the components leads to behaviour remaining compatible with existing, inter-dependent components, while possessing characteristics more suited to a certain input.

In designing diversity into collective systems, mutation and selection, as in biology, is indeed one potential approach. In evolutionary computation, for example, mutation is used to create diversity in a population of candidate solutions, which then compete in terms of their ability to solve a problem [19]. As will be discussed in section III-B, this genotypic/phenotypic diversity is also present in related techniques such as particle swarm optimisation. However, these algorithms typically restrict their use of diversity as a means of searching a solution space. They do not classically employ diversity of other characteristics, such as in terms of the search behaviour itself[1].

III. Diversity Case Studies

In this section we present three qualitatively different case studies, each in a different application domain, but in all of which diversity can play a positive role. These are i) distributed smart camera networks, ii) particle swarm optimisation, and iii) load balancing in a cloud computing environment. These case studies also vary in the degree to which differences in structure and properties pre-exist. In smart camera networks, the network structure and roles of components are predefined, based on camera location. In PSO, structure and roles emerge automatically and change over time in response to the problem and state of the collective. In contrast, the load balancing system does not function as a collective, instead serving as an example of temporal diversity in a single component system.

A. Smart Camera Networks

Smart cameras are fully computationally capable devices endowed with a visual sensor, and typically run computer vision algorithms to analyse captured images. Where standard cameras can only provide plain images and videos, smart cameras can process these videos to provide aggregated data and logical information, such as the presence of an object of interest. Communication between cameras allows a network of them track objects in a distributed fashion, handing over object tracking responsibilities from camera to camera as objects move through the environment. The approach studied here is that proposed by Esterle et al. [20], in which cameras exchange responsibilities through auctions, sending auction invitations to other cameras, who may then bid to buy objects. Performance at the global level is of primary interest, and consists of two network-level objectives: *tracking confidence*, a measure representing the ability of the network to track objects well, which should be maximised, and the *number of auction invitations*, a proxy for communication and processing overhead, which should be minimised. An outcome which

[1] A notable, fairly simple, but surprisingly effective exception is the self-adaptive mutation employed by evolution strategies [19]. More modern meta-heuristics also often incorporate diversity (typically referred to as heterogeneity) for other purposes, e.g. those PSO variants surveyed in [14], as will be discussed in section III-B.

achieves a high tracking confidence and a low number of auctions simultaneously can be described as being efficient.

The cameras use pheromone-based online learning to determine which other cameras they trade with most often. This neighbourhood relationship graph enables them to selectively target their auction invitations and achieve higher levels of efficiency, by avoiding communication with irrelevant cameras. In an extension to this work [13], six different behavioural strategies were available to cameras, which determined the level of marketing activity undertaken, given the vision graph. Some strategies incurred higher levels of communication overhead but typically obtained higher levels of tracking confidence; other strategies obtained the opposite results. However, the trade-off realised by each strategy was found to be highly dependent on the scenario and the properties of the camera in question; as camera positions and object movements varied, so did the relative benefits of the strategies.

Lewis et al. [13] showed that due to varying camera properties, heterogeneous behaviour configurations can lead to increased network-level tracking performance while simultaneously decreasing the number of auction invitations. Heterogeneity led to more Pareto efficient outcomes than were possible with homogeneous configurations. Furthermore, cameras were then each endowed with an independent reinforcement learning algorithm to select between behaviours from the available pool at runtime, thus changing the cameras' behaviours over time. These behavioural dynamics led, in many cases, to outcomes which were more efficient even when compared to the best possible outcomes from an exhaustive search of possible static heterogeneous configurations. Not only did behavioural heterogeneity between cameras enable more efficient outcomes, but behavioural dynamics led to a further efficiency increase. Favourable outcomes were obtained, which could not be reached in the static heterogeneous case. Clearly, both forms of diversity, namely heterogeneity and dynamics, provided benefits.

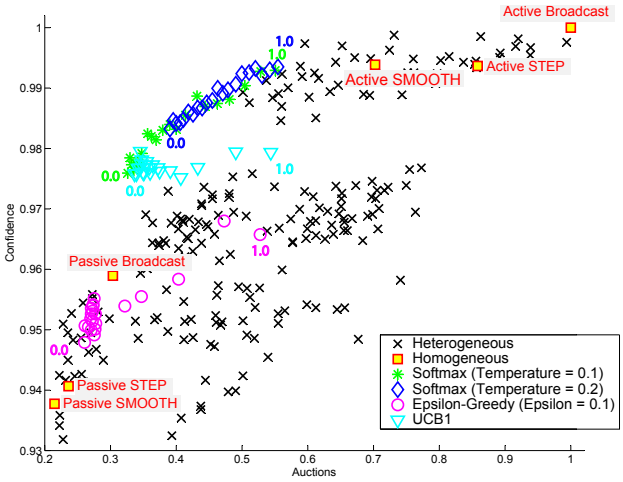

Fig. 1. Results for a typical smart camera scenario. Outcomes are shown for homogeneous, heterogeneous and learnt (dynamic) behaviours. Higher confidence and a auction rate is better. Heterogeneity provides more possible outcomes; many are more efficient than homogeneous ones. Dynamic behaviour provides consistently efficient outcomes. Source: [13].

Figure 1 shows outcomes achieved by homogeneous, heterogeneous and dynamic behaviours in a typical scenario. The results described here were observed on a wide range of scenarios [13], using an open source simulation package and real video feeds.

B. PSO

Particle Swarm Optimisation (PSO) is a population-based optimisation algorithm in which each particle attempts to minimise the value of some cost function, on the basis of its own observation of the function's values and selected observations (historic best positions) communicated to it by neighbouring particles. Each particle should behave in such a way as to maximise the utility of this knowledge. There are many variants of PSO behaviour with different performance profiles across problem types and, in general, predicting which single, static, behaviour will perform best for a given problem is difficult. Recent work has shown that allowing particles to randomly and independently switch between a set of behaviours is not only robust, but can outperform both static heterogeneous strategies and those which use simple feedback strategies to adapt to their input [14].

It is interesting to note that, in this example, input diversity does not have a purely external source as the problems used in the study were static; instead, input diversity is a direct consequence of diversity of particle position. The initial heterogeneity of position and subsequent heterogeneity of knowledge motivates heterogeneous behaviour.

PSO can also be characterised as semi-structured: while swarms have some defined topology to determine how particles communicate, particles are typically behaviourally equivalent. In most variants, roles emerge in response to the environment: particles with higher quality information about the search space become "leaders" and drive the search behaviour of the swarm. A particle's knowledge and role changes so rapidly that static heterogeneity is not appropriate. While in principle adapting based on input would be preferable, the speed and unpredictability with which a particle's input changes makes doing this well impractical, and renders stochastic choice a good practical solution.

C. Load Balancing

Load balancing for web-servers is the process of redirecting traffic from clients to servers, such that no particular server is overwhelmed. Depending on the characteristics of the traffic experienced, many algorithms are available. For example, the *round-robin* algorithm multiplexes traffic from clients among a pool of servers, using a fixed ordering. A *source-hashing* algorithm creates a hash of the source address, and allocates traffic based on which server services that bucket of hashes. This is to ensure that stateful web-applications always connect to the same server. Each algorithm has a different notion of the kind of traffic experienced and, therefore, exhibits good performance in some particular situations. However, for a globally accessed website, the pattern of traffic can be random. There is no known algorithm that can produce the best performance across a range of traffic patterns. We introduce diversity into the algorithm being used by the load balancer, and show that adding diversity can decrease failure rates. Instead of creating a new algorithm that deals with an unknown pattern of traffic, we

modify the load balancer to rotate the algorithm being used, every 20 seconds. Thus, any assumptions about the pattern of traffic lasts only for 20 seconds. Depending on the actual pattern of traffic, this will result either in successful handling of client requests, or failure indicated by the unavailability of a server.

We use the widely used `haproxy` (`http://www.haproxy.org`) as our load balancer. We measure the percentage of failure experienced by clients when the load balancer is hit by a high and randomly changing pattern of traffic. To allow for a sufficient 'ramp-up' time, we measure the failure rate over 1.5 million requests, with each load generator sending between 800-1000 requests per second. We divided the algorithms into various classes, based on combination size. Each of the individual algorithms was placed in class A. A pool consisting of triplets of randomly chosen algorithms was called class B, i.e. class B contained several combinations of three algorithms (*e.g.*, hdrHost-roundrobin-leastconn form a triplet. `haproxy` cycles between these three algorithms during a load testing run). Class C was a pool consisting of combinations of four randomly chosen algorithms. Finally, a pool containing all seven algorithms was called class D.

Figure 2 shows the failure rate when class A was used in a load testing run. It is clear that the header-based hashing algorithm had the lowest failure rate. Its worst performance (17% failure) is better than the next best algorithm (leastconn). Next, we ran the load test against random pools from each of the other classes (B, C, D). In Figure 3, we see the failure rates across classes. As expected, in class A there is a wide variation in the failure rates across various algorithms. Classes B, C, and D exhibit a lower median failure rate. However, class C also exhibits a much greater variation; all pools in class C are not equally effective. The graphs show the difference in failure rate when haproxy runs with any single algorithm, and when it cycles between multiple algorithms. Using a mixture of algorithms leads to a lower failure rate.

This problem contains *temporal diversity* in the input, from two sources: (1) the pattern of traffic (2) the kinds of workload. The diversity in the solution domain comes from two sources: (1) the natural diversity of capacity (speed, RAM, cache size, etc.) amongst application servers (2) the natural diversity of algorithms used by the load balancer. Typically, both these diversities are introduced to deal with input diversity of the second type, *i.e.*, diversity of workload.

The input diversity of unpredictable traffic patterns is not handled by the natural diversity in the solution domain. Creating a pool of algorithms, that `haproxy` cycles through, generates behavioural diversity in the load balancing system. In the load balancing scenario, minimising failure and lowering response times are key goals. Introducing behavioural diversity aids in achieving the first goal, without sacrificing the second. However, switching between algorithms to induce behavioural diversity introduces an engineering cost into building and maintaining a system. Not only must a diverse set of algorithms be available in the first place, their combined effect must also be understood. Diversification cannot always guarantee an improvement; not all diversity is good for the robustness of the system. Rather, the introduction of diversity is a tactic that

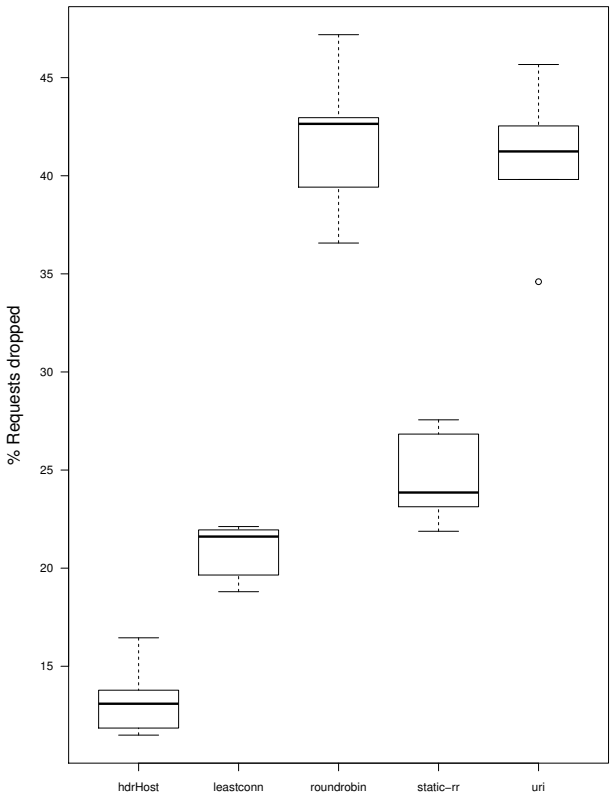

Fig. 2. Failure rate of single algorithms used by Haproxy.

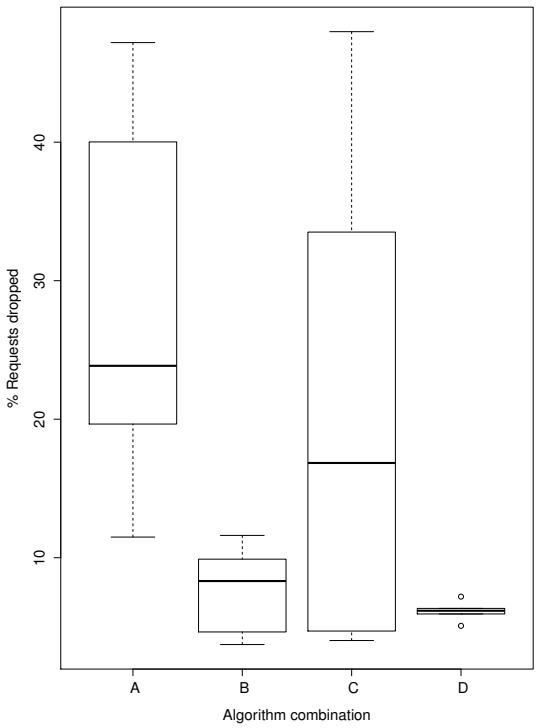

Fig. 3. Failure rates across all classes.

can provide measurable benefits, provided it is engineered in effectively, and not randomly chosen.

D. Common Observations

In the smart camera and the PSO case studies, we see that heterogeneity of system properties leads to heterogeneity of input and that this motivates behavioural heterogeneity. In both cases, heterogeneity alone can have a positive effect on performance. However, the stability of the system properties on which input depends determines the time at which we can make effective choices about behavioural diversity. If a property upon which input depends is stable (as with the position of a camera), then we can make decisions about introducing heterogeneity at design time. This may not be sufficient to design optimal behaviour, but allows for improvement upon a static homogeneous solution. Where a property upon which input depends is unstable (as with a particle's position in PSO), a static assignment of behaviours at design time is unlikely to be an optimal solution: rapid changes imply that dynamics should also be taken into account.

Each case study exhibits input dynamics, for different reasons. In principle, if we have a series of "specialist" behaviours and can either predict for, or adapt to input diversity such that each input is handled by an appropriate behaviour, then our system should, at worst, perform at least as well as one using a single "generalist" behaviour capable of handling a wide range of inputs. However, in the cases studied, a principled method of predicting or adapting to input dynamics was not known. Instead, behavioural dynamics have a positive effect on performance, filling this gap. We observe that switching between behaviours can allow a system to benefit from positive aspects of behaviours while mitigating negative ones. This is a powerful design tool for those problems in which potential inputs differ so greatly that a good generalist behaviour is prohibitively challenging to design.

IV. Conclusions and Future Directions

In this paper, we have argued that there are multiple types of diversity that can be inserted into computer systems. Genotypic/phenotypic diversity between components in a collective system or *heterogeneity*, and temporal diversity or *dynamics* are two such types. Each has a different effect on system behaviour and adaptive capacity, and each aids the ability to deal with the respective form of input diversity in the problem domain. Introducing such diversity in an effective way leads to a system being better adapted to varying problem spaces. A key observation of this paper is that adding diversity to the solution domain works as an effective tactic across qualitatively different case studies in this context.

While it is perhaps easy to acknowledge the need for diversity in systems, it is a far more difficult problem to design for effective diversity. From the load balancing case, it can be seen that adding diversity randomly may not always lead to successful outcomes. However, deciding *a priori* how much diversity to add, where and how it must be added and what are the measurable benefits and costs of adding diversity are all difficult questions. Therefore, more research is needed to determine how diversity, as a design characteristic, can be utilised in a principled manner to help achieve various quality attributes such as robustness, performance, resilience, adaptability etc. As part of this, how to design in or for optimal degeneracy, both between components and over time, will be an important question. An essential step in this direction would be to extend this work with a system-agnostic way of describing, both quantitatively and qualitatively, the heterogeneity and dynamics present.

Finally, in this paper we have concerned ourselves with cases when a designer can be assumed to have control over the diversity present in a system. Our conclusions are therefore important for techniques such as population based algorithms, as well as in applications such as sensor networks and the cloud. As the complexity and openness of systems grows however, designers will undoubtedly have less direct control over characteristics such as diversity. The distinctions presented here are also relevant to such cases, though methods will need to be developed by which such open systems can be steered towards exhibiting desirable diversity properties, despite their inherent decentralised control.

References

[1] A. Avizienis, "The n-version approach to fault-tolerant software," *Software Engineering, IEEE Transactions on*, no. 12, pp. 1491–1501, 1985.

[2] A. Avizienis and J. P. Kelly, "Fault tolerance by design diversity: Concepts and experiments," *Computer*, vol. 17, no. 8, pp. 67–80, 1984.

[3] T. Schnier and X. Yao, "Using negative correlation to evolve fault-tolerant circuits," in *In Proceedings of the 5th International Conference on Evolvable Systems: From Biology to Hardware (ICES2003), Lecture Notes in Computer Science*, vol. 2606. Springer, 2003, pp. 35–46.

[4] R. Feldt, "Generating diverse software versions with genetic programming: an experimental study," *IEE Proceedings-Software*, vol. 145, no. 6, pp. 228–236, 1998.

[5] S. Forrest, A. Somayaji, and D. Ackley, "Building diverse computer systems," in *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, ser. HOTOS '97. Washington, DC, USA: IEEE Computer Society, 1997.

[6] M. C. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, "Enhancing server availability and security through failure-oblivious computing," in *Proc. of the Operating System Design and Implementation (OSDI)*, 2004, pp. 303–316.

[7] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi, "Randomized instruction set emulation to disrupt binary code injection attacks," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, ser. CCS '03. New York, NY, USA: ACM, 2003, pp. 281–289.

[8] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a board range of memory error exploits," in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, ser. SSYM'03. Berkeley, CA, USA: USENIX Association, 2003, pp. 105–120.

[9] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 2003, pp. 272–280.

[10] L. L. Minku and X. Yao, "DDD: A new ensemble approach for dealing with concept drift," *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, no. 4, pp. 619–633.

[11] A. Campbell, C. Riggs, and A. Wu, "On the impact of variation on self-organizing systems," in *Self-Adaptive and Self-Organizing Systems (SASO), 2011 Fifth IEEE International Conference on*, Oct 2011, pp. 119–128.

[12] A. Prasath, A. Venuturumilli, A. Ranganathan, and A. Minai, *Sensor Networks: Where Theory Meets Practice*. Springer Berlin Heidelberg, 2009, ch. Self-Organization of Sensor Networks with Heterogeneous Connectivity, pp. 39–59.

[13] P. R. Lewis, L. Esterle, A. Chandra, B. Rinner, and X. Yao, "Learning to be different: Heterogeneity and efficiency in distributed smart camera networks," in *7th IEEE Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. IEEE Press, 2013, pp. 209–218.

[14] H. Goldingay and P. R. Lewis, "Taxonomy of heterogeneity and dynamics in particle swarm optimisation," in *Parallel Problem Solving from Nature - PPSN XIII, 13th International Conference, Ljubljana, Slovenia, September 13-17, 2014, Proceedings.*, To appear.

[15] N. Lathia, S. Hailes, L. Capra, and X. Amatriain, "Temporal diversity in recommender systems," in *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '10. New York, NY, USA: ACM, 2010, pp. 210–217.

[16] K. Schweikert, J. E. Sutherland, D. J. Burritt, and C. L. Hurd, "Analysis of Spatial and Temporal Diversity and Distribution of Porphyra (Rhodophyta) in Southeastern New Zealand Supported by the Use of Molecular Tools," *Journal of Phycology*, vol. 48, no. 3, pp. 530–538, 2012.

[17] C. Lévêque and J.-C. Mounolou, *Biodiversity*. Chicester, UK: John Wiley and Sons, 2003.

[18] C.-C. Chen and N. Crilly, "Modularity, redundancy and degeneracy: cross-domain perspectives on key design principles," in *8th Annual IEEE International Systems Conference*. IEEE Press, 2014.

[19] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing*. Heidelberg: Springer Verlag, 2003.

[20] L. Esterle, P. R. Lewis, X. Yao, and B. Rinner, "Socio-economic vision graph generation and handover in distributed smart camera networks," *ACM Transactions on Sensor Networks*, vol. 10, no. 2, 2014.