Losing Control: The Case for Emergent Software Systems using Autonomous Assembly, Perception and Learning

Barry Porter and Roberto Rodrigues Filho
School of Computing and Communications
Lancaster University
Lancaster, UK
Email: {b.f.porter, r.rodriguesfilho}@lancaster.ac.uk

***Abstract*—Architectural self-organisation, in which different configurations of software modules are dynamically assembled based on the current context, has been shown to be an effective way for software to self-optimise over time. Current approaches to this rely heavily on human-led definitions: models, policies and processes to control how self-organisation works. We present the case for a paradigm shift to fully *emergent* computer software which places the burden of understanding entirely into the hands of software itself. These systems are autonomously assembled at runtime from discovered constituent parts and their internal health and external deployment environment continually monitored. An online, unsupervised learning system then uses runtime adaptation to explore alternative system assemblies and locate optimal solutions. Based on our experience to date, we define the problem space of emergent software, and we present a working case study of an emergent web server. Our results demonstrate two aspects of the problem space for this case study: that different assemblies of behaviour are optimal in different deployment environment conditions; and that these assemblies can be autonomously learned from generalised perception data while the system is online.**

1. Introduction

Modern software systems are increasingly complex, and are deployed into increasingly dynamic environments. In recent years this trend has driven research in autonomic, self-adaptive and self-organising software systems [1], [2], [3]; this aims to move selected responsibility for system management into the software itself, thereby reducing the burden of complexity on human developers or administrators, and increasing responsiveness to dynamic environments.

One promising approach in this domain is the use of self-adaptive runtime software architectures, in which different software components are dynamically composed into the running system according to its current context. In these systems, there are multiple valid configurations of components that form a working system, but particular configurations perform better or worse in different deployment conditions. Recent examples of this approach include [4], [5], [6], [7].

In state of the art work, however, the way in which these systems are orchestrated relies on various forms of explicit control: models that describe adaptation states; architecture description extensions to specify autonomy; and policies to express runtime choices. We argue that these approaches continue to require substantial, detailed understanding of systems by humans, so that corresponding control strategies can be specified. This requirement is fundamentally opposed to the core ideas behind autonomic computing, which are borne of the increasing difficulty for humans to understand modern software systems in dynamic environments.

We present a novel approach to online software assembly, in which the understanding and control of that assembly is pushed deeply into software itself. We assemble software from a diverse palette of small building blocks that each provide a different element of functionality, where each such block has a pool of available *micro-variations* (the same behaviour implemented differently, such as memory cache components with different replacement algorithms, or stream processors that do or do not make use of a caching component). We then continually experiment with, and perceive the effectiveness of, different assemblies in various runtime conditions; we then see emergent designs of software appearing over time as external stimuli change.

Rather than focusing on human modeling of autonomy, we thus **enable machines to develop their own models, understanding, and methods of control**. By losing human control, we lose the ability to understand exactly how a given system works; what we gain, however, are software systems that are truly responsive to their environments, including the completely unexpected. In other words, instead of explicitly *programming* software in how to behave, and rather empowering it to *learn* how to behave, we achieve a level of freedom for systems to locate their own solutions in any conditions. We present two contributions to this goal:

- **A definition of emergent software systems**, in which software is abstracted into the key elements needed for machines to model, understand and control it. To test the limits of this we take a pure approach in which everything is learned by the system from experience, even if occasional mistakes are made in doing so. By giving away control to this extent, we correspondingly gain maximum responsiveness of software to actual deployment conditions – including intelligent responses to the completely unexpected, with no explicit programming to do so.

- **A framework to orchestrate emergent software**, using pluggable units for assembly, perception and learning. We present our experience to date on using this framework in the context of an emergent web server, a system that is autonomously assembled from small, dynamically-discovered component parts. We evaluate the resulting responsiveness of this system in a range of different conditions.

Our work is the first to examine complete behavioural emergence, learning how to form systems from small components, in which no prior models or policies exist. This paves the way to significantly reducing human involvement in software development, and increasing responsiveness of software to the actual conditions encountered at runtime.

Our work is also strongly grounded in reality: our prototype implementation is available as open-source software and comprises over 3,000 lines of well-modularised and extensible code, including a real working emergent web server system. The specific version used for this paper is available at [8], complete with instructions on how to reproduce all of the experimental results reported here.

The remainder of this paper is structured as follows. In Sec. 2 we discuss related work, and in Sec. 3 we present our definition of emergent software and our corresponding framework design and implementation. In Sec. 4 we then evaluate the system's ability to continually assemble optimal software as external stimuli change. We conclude in Sec. 5.

2. Related Work

While autonomic, self-adaptive and self-organising computing are now well established, there is relatively little work in runtime software composition (compared to far more work on autonomous parameter tuning). The majority of this work is model-driven, relying either on substantial human-specification; offline training regimes with historical data; or simple online heuristic search algorithms within a specified model. We survey the most closely related work here.

In [9], Grace et al. propose human-specified adaptation policies to select between different communication interfaces in a flood monitoring scenario. While the use of such policies is viable in simpler systems, this becomes infeasible in more complex systems where the set of component interactions is much larger. By contrast we use an online learning approach to *discover* an adaptation policy at runtime.

In [5], Chen et al. propose a weighted decision graph of service levels to generate model transformations in an online shopping system. Wang et al. [10], meanwhile, propose a framework that exploits variability of software configurations to deliver self-repair capabilities through reconfiguration, using a goal model based on requirements to drive this reconfiguration. We opt for a model-free approach in which components generate their own current service levels from which we infer global properties – an approach that reduces the burden of complexity on humans by avoiding the need to specify the initial system models and associated parameters.

In [11], Bencomo et al. propose dynamic decision networks (a form of state machine), alongside a models@runtime approach to software construction, to decide at runtime between different network topologies for a remote data mirroring system based on resilience levels. This requires pre-specification of the decision network to model configuration options, rather than the online learning approach we take for emergent software. In [12] the same authors use this example system to explore Bayesian prior/posterior differences as a trigger for when adaptation policies (encoded in dynamic decision networks built for the target system) do not match online experience. Our approach differs by building a model of understanding of the target software from scratch and at runtime, starting from no information, by assembling software from a pool of available building blocks and learning their characteristics.

In [13], Hassan and Bencomo use probability functions with Pareto analysis as a design-time tool in the software development process to help understand potential adaptation cases. Again, we avoid this need by using emergence from a set of building blocks, the characteristics of which are learned online according to the actual experience of the software in its deployment. By using small building blocks of relatively general behaviour we are able to view the adaptation problem as one of continuously forming beliefs from online learning to emerge a system, rather than discretely specified points of adaptation as part of a design process.

In [7], Kouchnarenko and Weber propose temporally-dependent logic to control software configuration, with a domain specific notation to model temporal dependencies between reconfiguration actions, using a self-driving vehicle control system as a case study. While the inclusion of such temporal models may be a useful addition for constraining adaptation, the models are again specified by human developers at design time rather than learned at runtime.

In FUSION [4], a feature-model framework is presented that uses offline training combined with online tuning to activate and deactivate selected feature modules at runtime (such as security or logging). Dynamic Software Product Lines [14] generalise the feature model approach as part of the software development process, typically using a pre-specified set of rules to trigger feature activation / deactivation at runtime. Our approach does not use a feature model, instead self-organising a pool of components into a working system; additionally we use pure online learning to make decisions, avoiding offline training or pre-crafted rules.

In SASSY [15], a self-adaptive architecture framework for service-oriented software is presented, using a set of model-based notations to describe software architecture and its quality of service traits. Further work by Ewing and Menascé [6] applies runtime heuristic search algorithms within these models to locate optimal configurations. Our work differs in using a model-free approach to emergent software, in which system composition is autonomously driven by discovering and experimenting with usable components and perceiving their contribution to the system.

The idea of 'organic computing' assumes scenarios in which many identical agents need to self-organise. Work in this area has attempted to define a theoretical framework to measure emergent behaviour [16], and to define a framework

for incrementally adding autonomous control to a given system [2]. Our work is fundamentally different to this in seeking emergent design of individual software systems through their autonomous assembly from many small (and different) building blocks to form desired behaviour.

Multi-agent systems [17] are a broader category of work similar to organic computing, again based upon many identical or similar agents acting independently (but able to communicate with one another) to achieve a macro-level goal that is more complex than the individual behaviour of any one agent. Our work can be positioned as the emergent design of an individual agent, given many options from which its behaviour can be assembled. As a result, our systems have a clear goal (such as 'be a web server') but the way in which they achieve that goal – their composition of behaviour, or their 'design' – is the emergent property for each environment in which the system finds itself.

We seek a radically different approach to all of the above by pushing understanding and control deep into software itself: we provide a sandbox of possibilities from which systems can emerge, and we expect particular designs to be learned for each environment that is encountered.

3. Emergent Software Systems

In this section we first define our concept of emergent software systems and the major challenges that they entail, based on our experience of building these systems. We then present our implementation of a framework to realise our approach, using an emergent web server as an example.

3.1. Problem definition

We define emergent software systems as follows. There exists a goal G that is expressed in a particular form (goal definition is beyond our scope here). A set of small software units SU exists that can be composed together into systems to achieve this goal, where each $u \in SU$ has one or more behavioural *variations* (implementations that offer the same functionality but using different techniques). One or more u emits a stream of 'metrics' describing the current health of u, and one or more u emits a stream of 'events' describing the software's current external stimuli (i.e. inputs being received or deployment environment characteristics).

The aim of an emergent software system is then to continually maximise its satisfaction of G by assembling the most optimal collection of u (where the satisfaction degree is a result of the combined health of all selected u as reported by metrics) in each set of deployment environment conditions which the software finds itself at runtime (as reported by events). The membership of SU may change dynamically, and a system should be able to emerge effectively with no prior information about its goal, the population of SU, or the set of environments to which it may be subjected. To push this approach to its limits we assume that all activities undertaken by an emergent software system occur on the 'live' system, in that system's normal production environment, such that it can learn from what actually happens to the real system in execution.

The challenges involved in achieving this primarily relate to the way that we design the learning systems that orchestrate emergence – i.e. the way in which a system builds its own self-understanding and correspondingly controls itself. This includes the existence of divergent optimality, relative and moving performance baselines, autonomous abstraction of the environment, learning techniques and challenges, and implications for the software design process.

3.1.1. Divergent optimality. To move toward optimality in an emergent software system, the application domain should be such that the behavioural variations of each u offer differing levels of performance in response to different external stimuli (input data or deployment environment conditions). When metrics or events are offered by a given u, the same metrics and events should be offered by each variant of u to serve as an equivalent basis for comparison.

3.1.2. Everything is relative. Performance differences of various available compositions of behaviour are all relative: there is no baseline at the point of system inception. The system must therefore construct its own moving baseline from its own observations, where the benchmark of what is 'good' is updated whenever something better is found.

Additionally, what is 'good' under one set of external stimuli may be different under other sets. These stimuli must therefore be characterised as and when they occur, so moving performance baselines can be kept for each case.

3.1.3. Abstracting the environment. In order to measure the effectiveness of different compositions in different operating environment ranges as stated above, we must first be able to characterise the features of each such range as and when they occur. The optimal composition of behaviours for each operating range can then be determined. Online feature extraction is a difficult problem in machine learning [18], and a further problem is that we do not know in advance what ranges of values we may encounter and therefore how best to define the boundaries between each detected environment range. Additionally, the way in which environment ranges are detected should avoid the classic control loop problem of oscillation between two nearby choices.

3.1.4. Online and offline learning. The key requirement of learning for emergent software is that the system must learn according to what it actually experiences. The main problem with this is that a system may experience different external stimuli erratically, making it difficult to draw comparisons between different possible compositions of behaviour under consistent external conditions. In this context there are two main methods by which an emergent system can learn.

One is to perform online experimentation, where the live, running system is re-assembled into its different available compositions while executing, so that the relative performance of each such composition can be determined under the different external stimuli that are experienced. When the system observes environment conditions for which it does not have enough information on the behaviour of a particular software assembly, it may therefore trial that assembly to gather more data. When doing so, it is particularly important

to be sensitive to bad compositions as they have real effects; a simple approach here may be a sliding scale in which a composition is experimented with for an increasingly small amount of time proportional to how relatively 'bad' it is.

The other option is to perform offline experimentation, where the online system remains in its most optimal form as currently predicted by this experimentation. This requires external stimuli seen by the online system to be *repeatable* in offline experimentation. While capturing input patterns for this purpose may be viable (if potentially expensive), other characteristics such as available system memory or CPU loading experienced in the online system may be more difficult to replicate offline. Further, the use of such offline learning is not 'free' and comes with its own resource costs.

Hybrid solutions may also be possible which fuse combinations of online and offline learning – such as experimenting with individual components using different input ranges, rather than experimenting with entire system compositions.

3.1.5. Search space complexity. Whichever kind of learning is used, the size of the search space grows rapidly as more u variants are added to SU. This is a combinatorial problem as each additional u variant composes with many other possible u variants around it. This requires creative solutions to learning which, as a system grows in complexity, ideally avoid the need to exhaustively trial every possible combination of components. This is an open challenge that demands novel solutions – with at least one possible avenue here being to intelligently share learned information across different software compositions in cases where those compositions have some common elements from SU. This may help to avoid the need to test compositions that are (either heuristically or from prior experience) sufficiently similar to other compositions that they can be considered equivalent under particular external stimuli.

3.1.6. Self-referential fitness landscapes. As reported by Cakar et al. in the context of parametric self-optimisation [19], it is likely that changes to the currently chosen set of u that make up a running system are capable of impacting upon the external stimuli that the system experiences. As an example, the performance of two different variants of u may be compared when subjected to equal external stimuli. If we define this external stimulus as a stream of requests, one variant of u may be more efficient and thus cause a higher rate of requests to be serviced, changing the apparent request pattern and so making it difficult to compare both variants of u under the 'same' conditions (as each u changes those apparent conditions when it is used).

3.1.7. Propagating errors as degraded health. We are used to handling errors and catching exceptional behaviour in code by using appropriate constructs available in a programming language. In an emergent software system, however, the system itself must be able to learn from errors, especially when the existence of certain errors depends on the current environment context. As an example, imagine that we have a sorting algorithm implemented in a given u, and we have a variant of u which uses a GPU-accelerated algorithm instead the regular CPU-based implementation. If we experience a GPU failure, this would usually be an error propagated internally by the software system.

To propagate this to an emergent system controller, the metrics emitted by a given u must reflect the error – for example if metrics report sorting speed, this may mean reporting extremely slow sorting speeds so that the emergent system can detect the change in performance and re-learn an optimal composition of behaviours for that machine (in this case using the CPU-based version of u).

3.1.8. Unexpected properties. As reported by Fisch et al. [16], perhaps the ultimate aim of emergent software systems is to demonstrate the unexpected: that autonomous learning activities produce an unexpected solution to a problem that is more than the sum of the individual parts available. While this is of interest, we also note that it is useful for emergent software to locate designs that are unexpectedly good for a given set of external stimuli. This is different from 'unexpectedly complex behaviour' as we expect the overall behaviour of a system to match our goal. Instead this reflects finding of designs for that behaviour, from among available fragments of behaviour in SU, which are unexpectedly good and therefore lead to new design knowledge.

3.1.9. Developer interaction. Finally, we look beyond emergent software as 'finding good solutions to a goal' to state that they are a natural way to invert the software development paradigm: that emergent software should, based on its actual experience, be able to make suggestions to human developers (or even machine agents) for new units of SU to be generated for particular criteria of external stimuli. This leads to a process wherein software itself plays an active role in its own development, suggesting improvements and testing them out, and further reducing the burden of complexity on human developers in systems building.

3.2. Prototype and case study

To realise our approach we have developed a prototype emergent software framework, along with a case study of an emergent web server. We implement the building blocks of our web server using a runtime component model, where each component has a range of micro-variations. These variations are trivial to create because each component is itself very small: examples are different caches with various cache replacement algorithms, and stream handlers that do or do not use caching. More detail on this is given in Sec. 3.2.2. Our emergent software framework is then divided into three major modules: an **assembly** module, responsible for discovering and assembling / re-assembling the target system from available components; a **perception** module, responsible for perceiving the current wellbeing of the target system and the state of its operating environment; and a **learning** module, responsible for inferring correlations between the software's current assembly, its perceived wellbeing, and the perceived conditions of its operating environment. The learning module is also responsible for characterising the various conditions of the operating environment, and for balancing exploration of untested software compositions with exploitation of compositions known to perform well.

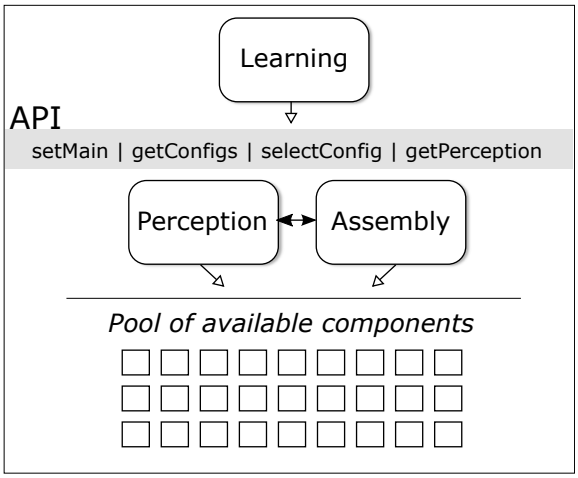

Figure 1. Architecture of the emergent software framework.

```
data Event {
   char name[]
   int value
}

data Metric {
   char name[]
   int value
   bool preferHighValue
}

interface Recorder {
   void addEvent(Event e)
   void addMetric(Metric m)
   PerceptionData getPerception()
}
```

Figure 2. The Event and Metric data types and the Recorder interface, using the syntax notation of the component model that we use [20].

Using the terminology of Sec. 3.1, our goal G is expressed in terms of a 'main component' that encodes the overall task of the web server and has a set of 'required interfaces' that express further components that are needed. A set of unit tests is available that can verify whether or not this component (and therefore the system as a whole) is delivering the expected functionality. Starting from this main component, our framework dynamically discovers all other possible components SU from which to build the rest of the system. Our framework then begins to experiment with these components to locate optimal assemblies of a system for each set of external stimuli that are experienced, perceiving any events and metrics that are emitted. Throughout this process a fully functioning web server is maintained.

We specifically use [20] as our runtime component platform due to its affinity for fine-grained components and very fast runtime adaptation. Other component models could also be used, however. In the remainder of this section we first describe our emergent software framework in detail, and then how we apply it to our emergent web server example.

3.2.1. Emergent software framework. Our emergent software framework is a generalised system capable of performing three main tasks: assembling a piece of software from a collection of available components, perceiving its performance and external stimuli at runtime, and learning about how that performance relates to its external stimuli.

These three elements, illustrated in Fig. 1, are arranged in two tiers. The perception and assembly modules sit at the lower tier and provide a simple API to the learning module at the upper tier, allowing the learning module to control and perceive complex software systems using simple primitives. We now describe the general role of each module in detail, along with the details of our learning module implementation as used in our evaluation in Sec. 4.

Assembly module This component is responsible for discovering and assembling the target systems's components. It is provided with the 'main' component of the target system and examines this component's set of required interfaces, scanning the local system for all components that declare matching provided interfaces. These components are themselves examined to discover *their* required interfaces, and so on, until the assembly module has a complete map of all possible compositions of the target system (including all available variants of each provided interface, for example different memory cache or ADT implementations).

The result of this process is a set of possible configurations of components. A unique string is assigned to each one so that they can be referred to by other parts of the framework. This string contains a (compressed) list of all components of the configuration and their interconnections. The list of configuration strings is accessed via the `getConfigs()` API call shown in Fig. 1.

The assembly module can then be instructed to adapt the target emergent system to one of these configurations using the `selectConfig()` operation. If the target system is not yet assembled, this simply involves loading all of the necessary components into memory, interconnecting them, and calling the 'main method' of the main component to start the system. If the target system is already assembled, this involves comparing the currently assembled configuration and the new target configuration to build a minimal differential graph between the two. Each point in that graph is then adapted by loading the alternative component, using the runtime component model's adaptation protocol to replace the existing component at that location with the new one, and then unloading the existing component at that location.

Perception module This component perceives the behaviour and performance of the currently assembled configuration, and the state of the system's operating environment.

To do this, our framework uses a `Recorder` interface, which is shown in Fig. 2. Any component can declare a required interface of this type, which will be connected to a corresponding component implementing the Recorder interface. The interface provides functions for components to log the emission of metrics and events when they occur.

Metrics are used to describe the way that the software 'feels', and have a standard format including a name, a value and a boolean flag indicating whether a high or low value is considered to be better. When a metric is logged at a

recorder, a timestamp is also added. An example metric would be 'response time', with a value in milliseconds, and a boolean flag indicating that lower is better in this case.

Events describe the way that the software's deployment environment 'looks', with a standard format including a name and a value. As with metrics, a timestamp is added when an event is logged at a recorder. An example event would be 'request', with a request type as a label, and a numerical characteristic (e.g. request size) as the value.

Whenever a new configuration of the target system is assembled, the perception module scans all of its components for any with a Recorder required interface. The Recorder component attached to each one is then periodically polled to collect the latest batch of events and metrics generated by the associated component. The `getPerception()` call can then be used on the perception module, returning a *PerceptionData* structure containing all events and metrics that have been collected along with their timestamps.

Learning module This component uses the API provided by the assembly and perception modules to experiment with, understand and control the target emergent system. This is done with no prior knowledge of what that target system is, or knowledge of the operating environment conditions that may occur (including no knowledge of what kinds of events / metrics may be emitted). The task of the learning module is to understand the correlations between the currently assembled collection of components (i.e. the software system's current behaviour), and how the system is currently perceived to be feeling, in each set of perceived operating environment conditions. The learning module is able to experiment with behavioural changes, by asking the assembly module to select a different configuration, to understand how different behaviours then affect the software's perception of self in different operating environments.

To implement this, the learning module has two main tasks. First, it must be able to characterise and classify features in the software's operating environment (derived from the stream of events being emitted) so that the performance of different configurations can be compared equivalent environments, and so that the learning module can remember which configurations work best in each environment (i.e. to save re-learning each time a recurring environment is encountered in the future). And second, as in any online learning system, the learning module must balance the trade-off between exploring options about which there is insufficient information and exploiting options known to be good [21]. Because our emergent software framework operates on live software, this balance is particularly important because operating sub-optimally has real consequences.

Performing both tasks online is highly challenging: the software is not in control of its current operating environment and so cannot know in advance when it may be able to reliably compare any two software configurations against the same set of external stimuli; in addition there are complex interactions between the process of exploration itself and the environment, where e.g. selecting a 'good' configuration can increase throughput and so change the perceived environment. We use an approach inspired by reinforcement learning [21], modified for our particular problem space. Our solution, shown in Algorithm 1, continually locates the optimal configuration by incrementally exploring the configuration search space while simultaneously characterising observed external stimuli into discrete labelled environments.

Algorithm 1 Learning Algorithm

```
1:  while running do
2:      //perform exploration activity
3:      for each c in assembly.getConfigs() do
4:          assembly.selectConfig(c)
5:          wait for w_t
6:          store perception.getPerception() for c
7:      end for
8:
9:      //select the new configuration to use
10:     store environment ep as max : min of event types
11:     assembly.selectConfig(best known for ep)
12:
13:     //wait for conditions to change
14:     newExploration = false
15:     while newExploration == false do
16:         wait until (different environment ep detected) or
17:                 (performance degrades) for >= w_t * 3
18:         if different ep and ep is previously known then
19:             assembly.selectConfig(best known for ep)
20:         else
21:             newExploration = true
22:         end if
23:     end while
24: end while
```

The algorithm uses a standard 'exploration activity' in which to both characterise the current environment and also identify the best configuration for that environment, shown on lines 3-6. The system triggers this exploration whenever it encounters sufficiently high uncertainty about its current choices – where this uncertainty comes either from (i) having no information at all (i.e. system startup); (ii) the current environment characteristics deviating outside of expected ranges from existing experience, or (iii) current system performance deviating beyond its expected range.

The exploration activity tries every possible configuration for a fixed-length 'observation window' w_t, such that the total time spent exploring is $w_t * length(getConfigs())$. We define w_t as 10 seconds for this paper. Having tried every configuration, the learning module then characterises what happened over the entire exploration time period to determine the best course of action as a result of that exploration activity. This characterisation works by considering all events and metrics that were reported during exploration, and for each distinct event type (qualified by having a unique event name) a max-min range is determined by extracting the minimum cumulative 'value' of this event type from all w_t within that exploration activity, and a maximum cumulative value of this event type from all w_t. The environment is then labelled as the set of these max-min ranges for all event types perceived during this exploration activity. The

best performing configuration within this environment is then chosen (line 11) as that with the best set of perceived metrics during its w_t; this configuration becomes the 'rule' for use whenever this environment is encountered.

The use of ranges to classify an environment addresses the self-referential fitness landscape issue (see Sec. 3.1), in which some configurations may be better and thus appear to alter their own environment by (for example) consuming more data at a higher rate – our ranges capture the highest and lowest levels of environment perceived during an exploration activity, abstracting over these details. The main problem with our approach comes when the environment changes significantly *during* an exploration activity, meaning that the various configurations used were not really compared in the same conditions. This issue is addressed by our second two uncertainty clauses (listed above), captured on lines 14-23.

Specifically, after an exploration activity, the learning module selects the best-performing configuration for use and enters its exploitation state. The selected action continues to be monitored every w_t and analysed for its suitability. A change may occur if either (i) perceived events during w_t show that this is a different event pattern (i.e. they fall outside the range of the current pattern), or (ii) perceived metrics during w_t show degraded performance. To avoid frequent oscillation, in either case the algorithm waits for $w_t * 3$ of consistently observed behaviour before changing its current course. In case (i), if the detected event pattern is one that has been previously seen, the matching best configuration is simply selected. In all other cases a new exploration activity is triggered. This process of exploration / exploitation repeats continually, where the amount of exploration will reduce as fewer new environments are seen.

3.2.2. An emergent web server. To test our framework we use a web server as an example emergent system. This is a pertinent example because web servers are known to be difficult to optimally configure, particularly when subjected to different client workloads over time [22]. In this section we first describe the main components of our web server and their available variants, and then we discuss the events and metrics that we chose to generate from these components.

Architecture The main components from which our web server can emerge are shown in Fig. 3, indicating the pool of behaviours from which our framework can choose to assemble a system. Note that the actual set of components used is much larger than this, including string utilities, file system and TCP socket APIs, abstract data type implementations, etc. Here we focus on the components that have variation.

There is a single main component which is passed to our framework to start the system. This component simply opens a TCP server socket and then accepts new client connections, passing each one to a 'request handler'. Our request handlers introduce concurrency and we have two variants: one that creates a new thread for every client, and one that uses a pool of threads on which to enqueue clients. From here a client is passed to a 'HTTP handler' which parses the request and forms a response. We have four variants of this component, which do or do not use caching / compression. We then have various versions of cache and compressor components for HTTP handlers that use them.

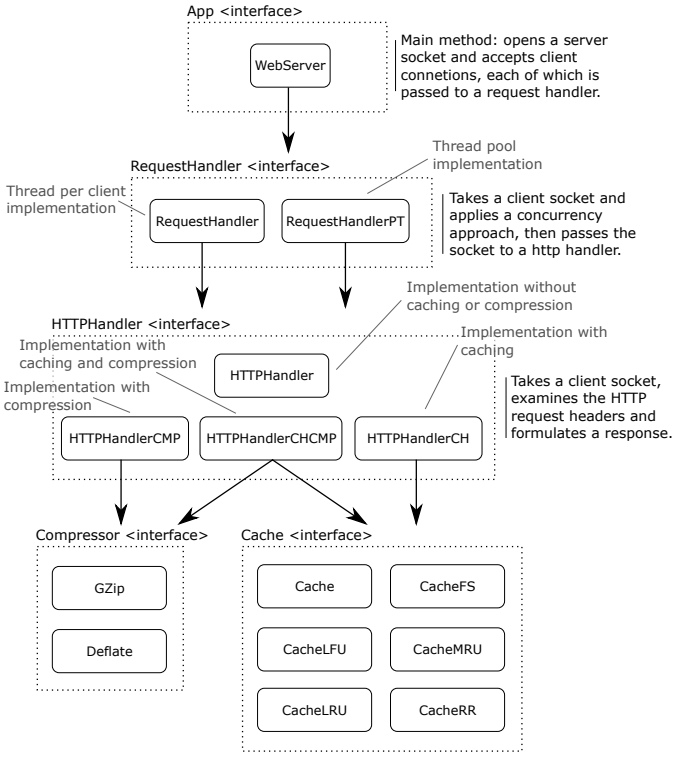

Figure 3. The set of components from which our web server can emerge. Boxes with dotted lines are interfaces, and those with solid lines are components implementing an interface. Arrows show required interfaces of particular components. The general purpose of each interface's implementations is noted by the interface, and a description of how the available implementation variations of that interface work is also indicated.

These components can be used to create a wide range of valid web server architectures at runtime, and the component population can be added to over time as new components become available. All such components, and the ways in which they can be combined, are dynamically discovered by our framework, and provide a wealth of micro-variation where different collections of behaviours may provide different levels of performance under different conditions. In total there are 42 possible configurations of our web server that can be formed from these components.

Events and metrics The events and metrics generated by these components determine how our emergent software framework perceives and understands the system and its environment, and learns to best control the system in operation. We currently use one metric type and one event type.

Our metric type is generated by all 'request handler' variants, and reports the total response time to each request. Our event type is generated by all 'http handler' variants, and reports the request types that arrive at the server and their sizes. Note that because we generate a lot of metrics, our Recorder implementation aggregates their values over time, storing only the sum of response times and the number of metrics that have been collected (allowing us to calculate an average without storing each individual metric).

4. Evaluation

In this section we evaluate our emergent software framework, using our web server as an example emergent system. Our evaluation was conducted with a real implementation of our web server, and our emergent software framework, running on rackmount servers in a production datacentre. These servers have an Intel Xeon E3-1280 v2 Quad Core 3.60 GHz CPU, 16 GB of RAM, and run Ubuntu 14.04. Similar specification machines were used as clients to generate workloads; these client machines were on a different subnet to the servers (in a different building). We use a mixture of custom-built workload patterns designed to explore our system's characteristics in targeted ways, and a real-world trace from NASA [23]. We demonstrate two key results:

First, we show that different web server configurations (i.e. different compositions of the available components) perform better in different operating environments. More specifically we show that there exist cases in which one configuration A is best in some environments, while another configuration B is best in others. We refer to this phenomena as 'divergent optimality', which motivates our approach.

Second, we show that our approach can correctly select the optimal configuration, from all those available, using only online learning and with no human input or prior knowledge. This occurs continually such that if the operating environment changes our platform will identify the optimal configuration for that new set of conditions.

All code used in our evaluation, along with instructions on how to repeat all of our experiments, is available at [8].

4.1. Divergent optimality

In this section we show how different web server configurations perform differently when subjected to different request patterns. Results from custom-defined request patterns are shown in Fig. 4–6; while results from the NASA trace are shown in Fig. 7. For these graphs we have selected four specific configurations, from the 42 available, that are most different in terms of the behaviour that they include.

Fig. 4 and Fig. 5 show the average response time of the web server for request patterns in which the same file is repeatedly requested. When this is a text file, Fig. 4 shows that configurations with in-memory caching and without compression have better average response times than configurations with both caching and compression. However, for image files, Fig. 5 shows that the opposite of this is true.

In contrast to this, Fig. 6 and Fig. 7 show the average response time of the web server for request patterns in which many different files are requested. In detail, Fig. 6 shows results from a custom request pattern in which each request is for a different small (∼3KB) text file; while Fig. 7 shows results from replaying the NASA trace (which also has a high degree of variation). In both of these graphs we see that the best configurations from Fig. 4 and Fig. 5 are in fact the worst two configurations for these request patterns.

This clearly demonstrates that different configurations of our web server will perform differently when subjected to different request patterns at runtime. In particular, request

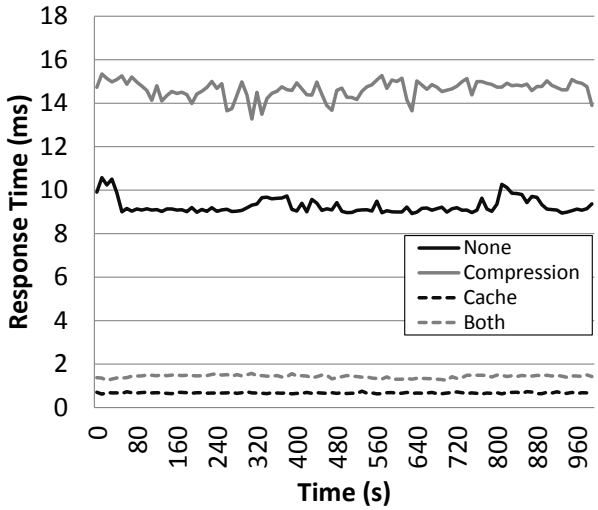

Figure 4. This graph illustrates the performance of four different configurations with the request pattern of small text files.

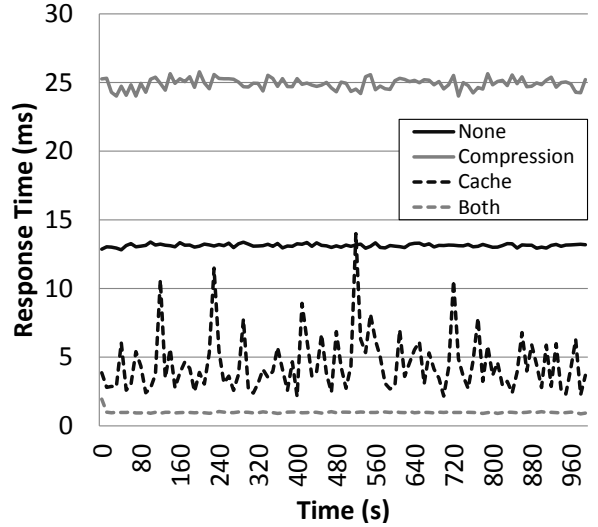

Figure 5. This graph illustrates the performance of four different configurations with the request pattern of small image files.

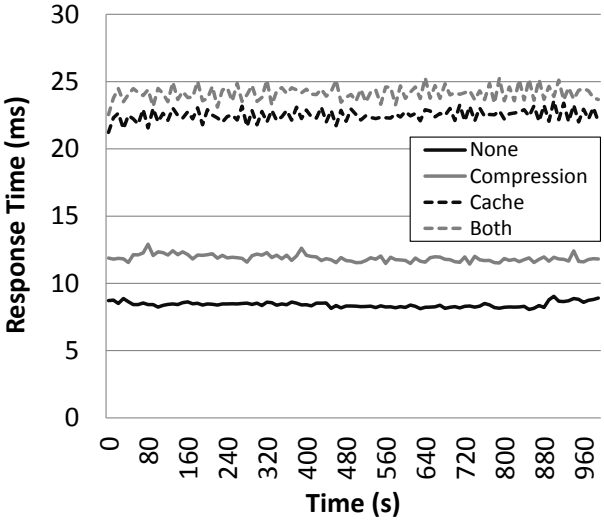

Figure 6. This graph illustrates the performance of four different configurations with the request pattern of a variation of small text files.

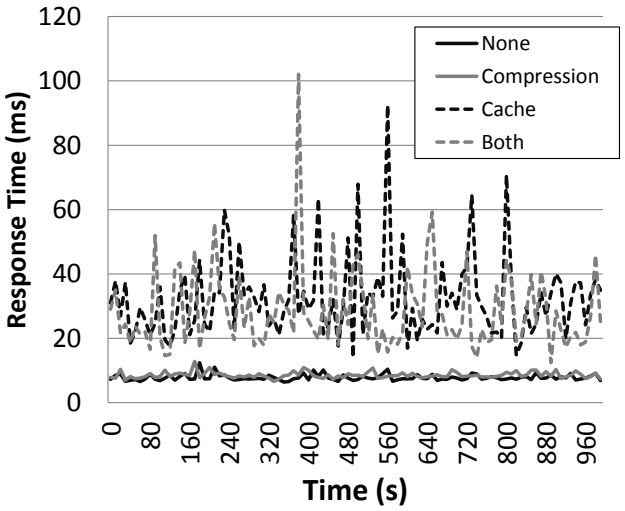

Figure 7. This graph illustrates the performance of four different configurations with the request pattern of the NASA trace [23].

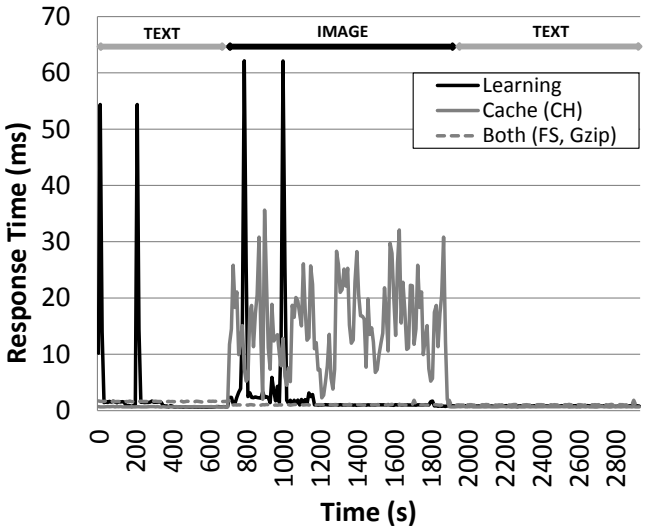

Figure 8. Performance comparison between fixed web server architectures and our emergent platform, using two different request patterns over time.

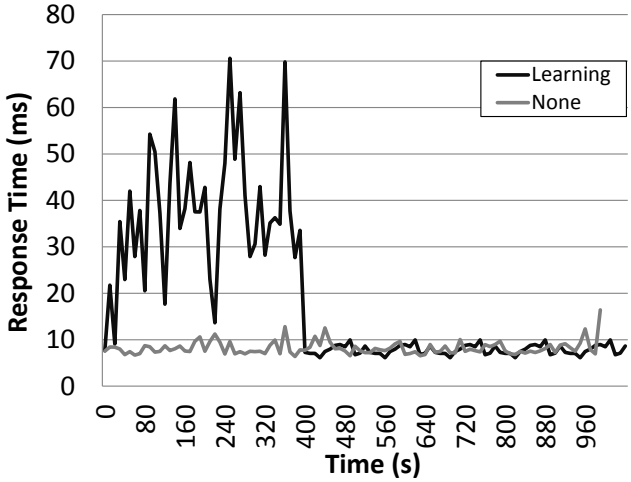

Figure 9. Performance comparison between a fixed web server architecture and our self-adaptive platform when using the NASA request pattern [23].

patterns with high variation do not benefit from configurations that use caching, whereas request patterns with low variation do. Additionally, the performance of architectures that include compression is impacted by the compression ratio of the files being requested in that pattern. While this may be intuitive to a human, in the next section we demonstrate the feasibility of autonomously learning this information from no prior experience – the basis of emergent software systems whose design is a product of their environment.

4.2. Online learning of emergent software

We now evaluate emergent software systems: continual, autonomous selection of the optimal component compositions for the web server, by analysing the currently available perception data (events and metrics) and exploring how the various available compositions of behaviour affect the perception of metrics across different environments. We achieve this using only unsupervised online learning, with no human input and with no application-specific aids.

Fig. 8 shows a request pattern consisting of sequential requests for small ($\sim$ 3KB) text files for 700 seconds, followed by sequential requests for small ($\sim$ 1MB) image files for 1200 seconds, and finally returning to small text files. This experiment was chosen as it contains two distinct kinds of request pattern for which different web server architectures are known to be optimal, as shown in Sec. 4.1.

The graph shows the performance of our online learning approach, exploring available compositions, compared to the performance of two different static web server configurations that are known to be optimal for the different phases of this request pattern. At the beginning of the experiment, the learning system starts with no information and so must go through the entire learning process to discover the architecture most suited for the currently observed conditions.

In detail, when a new pattern is detected, the learning module performs an exploration activity to find the best configuration for that pattern. This takes 420 seconds (each configuration runs for $w_t = 10$ seconds) and is clearly visible on the graph as two large spikes; each spike shows experimentation with a particularly poorly-performing configuration for this pattern. When learning is complete, our platform converges on the optimal configuration. This can be seen at two times, one at time 250, and the other at time 1200. At time 1900 we see another request pattern transition but, in this case, to a pattern that the learning system has already seen; this does not trigger a further learning phase and instead simply picks the best configuration from prior experience. Comparing this against the two static configurations we can see that our framework maintains optimal performance for the longest period of time, while both static configurations are optimal at some times but not others.

Fig. 9 shows an experiment with our learning system using the NASA trace, which is characterised by having small files ($<$ 20MB) with a high degree of variation, meaning that the same file is rarely requested consecutively.

This trace was chosen as a representation of a real-world scenario. Starting from no information at the beginning of the experiment, the learning process maintains the same time of 420 seconds to learn the most suitable configuration – again needing to experiment with each available configuration for 10 seconds. We compare this to the performance of a fixed architecture that had the best performance for this pattern, showing that after 420 seconds the learning system converges to an architecture with an equivalent level of performance. We note that, when compared to the results in Fig. 8, both the learning and static architectures in this case have a relatively erratic level of performance caused by a relatively high degree of variation in this request pattern.

These results demonstrate that, starting with no information at all, we can learn and converge on an optimal configuration in real-time. As more data is collected by the learning algorithm, more experience is gained, and less learning takes place – but the approach always maintains the ability to detect new conditions and react to them.

5. Conclusion

We have presented a definition of emergent software, based on our experience of implementing emergent systems, along with our framework for orchestrating emergent software and an example of a web server that exhibits these properties. From our definition of emergent software, our implementation demonstrates divergent optimality from different compositions (Sec. 3.1.1); addresses the issue that 'everything is relative' (Sec. 3.1.2) by implementing a moving baseline of optimality; presents a solution to abstracting the environment in real time (Sec. 3.1.3) using sets of quantified min-max event ranges; and uses a purely online approach to learning (Sec. 3.1.4) that takes into account the self-referential fitness landscape issue (Sec. 3.1.6).

In future work we plan to investigate further points in the design space for each of these concerns, as well as examining the topics of search space complexity, error propagation, unexpected properties and developer interaction. In addition we will explore further case studies of emergent software systems to help generalise our work to date – including distributed federations of locally emergent systems.

Acknowledgements

This work was supported by the UK's EPSRC in the *Deep Online Cognition* project, grant number EP/M029603/1. Roberto Rodrigues Filho would like to thank his sponsor, CAPES Brazil, for the scholarship grant BEX 13292/13-7.

References

[1] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 4, no. 2, p. 14, 2009.

[2] S. Tomforde, J. Hähner, and C. Müller-Schloer, "Incremental design of organic computing systems - moving system design from design-time to runtime," in *Proc. of the 10th International Conference on Informatics in Control, Automation and Robotics*, 2013, pp. 185–192.

[3] F. Faniyi, P. R. Lewis, R. Bahsoon, and X. Yao, "Architecting self-aware software systems," in *Proceedings of the IEEE/IFIP Conference on Software Architecture (WICSA)*, April 2014, pp. 91–94.

[4] A. Elkhodary, N. Esfahani, and S. Malek, "Fusion: A framework for engineering self-tuning self-adaptive software systems," in *Proc. of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2010, pp. 7–16.

[5] B. Chen, X. Peng, Y. Yu, B. Nuseibeh, and W. Zhao, "Self-adaptation through incremental generative model transformations at runtime," in *Proc. of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 676–687.

[6] J. M. Ewing and D. A. Menascé, "A meta-controller method for improving run-time self-architecting in SOA systems," in *Proc. of the 5th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '14. New York, NY, USA: ACM, 2014, pp. 173–184.

[7] O. Kouchnarenko and J.-F. Weber, "Adapting component-based systems at runtime via policies with temporal patterns," in *Formal Aspects of Component Software*. Springer, 2014, pp. 234–253.

[8] Source code from this paper with instructions: http://research.projectdana.com/saso2016rodrigues.

[9] P. Grace, D. Hughes, B. Porter, G. Blair, G. Coulson, and F. Taiani, "Experiences with open overlays: a middleware approach to network heterogeneity," in *Proc. of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, April 2008, pp. 123–136.

[10] Y. Wang and J. Mylopoulos, "Self-repair through reconfiguration: A requirements engineering approach," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2009, pp. 257–268.

[11] N. Bencomo, A. Belaggoun, and V. Issarny, "Dynamic decision networks for decision-making in self-adaptive systems: A case study," in *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2013 ICSE Workshop on*, May 2013, pp. 113–122.

[12] N. Bencomo and A. Belaggoun, "A world full of surprises: Bayesian theory of surprise to quantify degrees of uncertainty," in *Proc. of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014, pp. 460–463.

[13] S. Hassan, N. Bencomo, and R. Bahsoon, "Minimizing nasty surprises with better informed decision-making in self-adaptive systems," in *IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, May 2015, pp. 134–145.

[14] M. Hinchey, S. Park, and K. Schmid, "Building dynamic software product lines," *IEEE Computer*, vol. 45, no. 10, pp. 22–26, Oct 2012.

[15] D. Menasce, H. Gomaa, S. Malek, and J. Sousa, "SASSY: A Framework for Self-Architecting Service-Oriented Systems," *IEEE Software*, vol. 28, no. 6, pp. 78–85, Nov 2011.

[16] D. Fisch, M. Janicke, B. Sick, and C. Muller-Schloer, "Quantitative emergence – a refined approach based on divergence measures," in *Proceedings of the 4th IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, Sept 2010, pp. 94–103.

[17] J. Ferber, *Multi-agent systems: an introduction to distributed artificial intelligence*. Addison-Wesley Reading, 1999, vol. 1.

[18] K. Glocer, D. Eads, and J. Theiler, "Online feature selection for pixel classification," in *Proceedings of the 22nd International Conference on Machine Learning*, ser. ICML '05. New York, NY, USA: ACM, 2005, pp. 249–256.

[19] E. Cakar, S. Tomforde, and C. Müller-Schloer, "A role-based imitation algorithm for the optimisation in dynamic fitness landscapes," in *IEEE Symposium on Swarm Intelligence (SIS)*, April 2011, pp. 1–8.

[20] B. Porter, "Runtime modularity in complex structures: A component model for fine grained runtime adaptation," in *Component-Based Software Engineering*. ACM, June 2014, pp. 26–32.

[21] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*, ser. A Bradford book. Bradford Book, 1998.

[22] W. Zheng, R. Bianchini, and T. D. Nguyen, "Massconf: automatic configuration tuning by leveraging user community information," in *ACM SIGSOFT Software Engineering Notes*, vol. 36, no. 5. ACM, 2011, pp. 283–288.

[23] NASA web server trace: http://ita.ee.lbl.gov/html/contrib/nasa-http.html.