Defining Emergent Software using Continuous Self-Assembly, Perception and Learning

Roberto Rodrigues Filho, Lancaster Univeristy
Barry Porter, Lancaster Univeristy

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 continuously explore alternative system assemblies and locate optimal solutions. Based on our experience over the last three years, we define the problem space of emergent software and present a working case study of an emergent web server as a concrete example of the paradigm. Our results demonstrate two main 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.

ACM Reference Format:
Roberto Rodrigues Filho, Barry Porter, 2016. Defining Emergent Software using Continuous Self-Assembly, Perception and Learning. *ACM Trans. Autonom. Adapt. Syst.* V, N, Article A (January YYYY), 25 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Modern software is increasingly complex, and is deployed into increasingly dynamic environments. In recent years this trend has driven research in autonomic, self-adaptive and self-organising software systems [Salehie and Tahvildari 2009; Tomforde et al. 2013; Faniyi et al. 2014]; these research efforts aim to move selected responsibility for system management into the software itself, thereby reducing the burden of complexity on human developers or administrators, and increasing the responsiveness of software to dynamic deployment 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 from the research literature include [Elkhodary et al. 2010; Chen et al. 2014; Ewing and Menascé 2014; Kouchnarenko and Weber 2014].

In state of the art work, however, the way in which these self-adaptive systems are orchestrated relies on various forms of explicit control: models that describe adaptation

This work was partly supported by the UK's EPSRC in the *Deep Online Cognition* project, grant number EP/M029603/1, and by CAPES Brazil via PhD scholarship grant BEX 13292/13-7.
Author's addresses: B. Porter and R. Rodrigues-Filho, School of Computing and Communications, Lancaster University, Lancaster, UK.

DOI: http://dx.doi.org/10.1145/0000000.0000000

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 of *continuous online self-assembly* of software, 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 piece 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). While the system is running we continually experiment with, and perceive the effectiveness of, different possible assemblies of the system in various runtime conditions; as a result of this process we see emergent designs of software autonomously 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 to 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 and we discuss which aspects of our emergent software definition we have met in doing so.

Our work is the first to examine full behavioural emergence, learning how to form single systems from small components, with no prior models, policies or knowledge. This paves the way to significantly reducing human involvement in the complex details of software development, and to increasing software responsiveness to the actual conditions encountered by a system at runtime.

Our work is strongly grounded in reality: our prototype implementation is open-source and comprises over 3,000 lines of well-modularised and extensible code, including a working emergent web server system. The specific version used for this paper is at [Filho 2016], complete with instructions on how to reproduce all of the experimental results reported here. The original version of this paper was presented at IEEE SASO 2016 [Porter and Rodrigues Filho 2016]; in this extended version we provide an updated and more complete definition of emergent software as a concept, extended details on our prototype emergent software framework, and further evaluation results that consider concurrency, test our real-time classifier in detail, examine the effects of ob-

servation window sizes, and provide results from experiments on alternative hardware platforms which demonstrate autonomous learning of the host platform's strengths.

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 the paper and consider future work 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 [Grace et al. 2008], 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 [Chen et al. 2014], Chen et al. propose a weighted decision graph of service levels to generate model transformations in an online shopping system. Wang et al. [Wang and Mylopoulos 2009], 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 [Bencomo et al. 2013], 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 [Bencomo and Belaggoun 2014] 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 [Hassan et al. 2015], 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 [Kouchnarenko and Weber 2014], 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 ve-

hicle 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 [Elkhodary et al. 2010], 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 [Hinchey et al. 2012] 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 [Menasce et al. 2011], 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 [Ewing and Menascé 2014] 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 continually experimenting with usable components and perceiving their contribution to the system across different environments.

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 [Fisch et al. 2010], and to define a framework for incrementally adding autonomous control to a given system [Tomforde et al. 2013]. 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 [Ferber 1999] are a broader category of work with similar goals 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 and directly specified 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 design and implementation of a generalised 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). In our work to date,

each u is around 200 lines of source code, implementing concepts such as search algorithms or memory caches; this small size makes it relatively easy to create variations and yields a high number of possible combinations of those variations. We require one or more u to emit a stream of *metrics* describing selected aspects of the current health of u, and one or more u to emit a stream of *events* describing selected aspects of the software's current external stimuli (i.e. inputs being received or deployment environment characteristics such as the available remaining memory or current CPU load of the host system). This implies there is a fixed number of valid *assemblies* from SU that each result in a working system for G, where different assemblies perform differently.

The aim of an emergent software system is then to continually maximise its satisfaction of G by assembling the most optimal collection of u in each set of deployment environment conditions which the software finds itself at runtime, where the satisfaction degree is a result of the combined health of all selected u as reported by metrics and potentially weighted by non-functional qualifiers on the system goal, and the current deployment environment conditions are captured by the reported events. A functioning, optimal system should be able to emerge effectively with no prior information about its goal, nor about the population of SU or the runtime characteristics of any particular u, or the set of environments to which the system may be subjected at runtime. The population of SU is also expected to change dynamically, allowing the introduction of new behaviours at any time.

We assume that all activities undertaken by an emergent software system occur on the 'live' system, while that system is operating in its normal production environment, such that it learns from what actually happens in execution. As such, we assume that there exist automated unit tests by which any assembly candidate from SU can first be autonomously verified as being functional according to the system goal before such an assembly becomes a valid option for the live emergent system to use at runtime.

The challenges involved in achieving emergent software 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 (i.e., different input data or deployment environment conditions). Depending on the environment conditions actually observed by a system over time, this may mean that the emergent system either finds one overall optimal solution from SU and uses that solution permanently, or that it finds different optimal solutions to match different environment conditions as they are encountered. In either case, the emergent system continually remains ready to react to the unexpected, learning new environments and new ways of assembling the system as and when they become apparent.

To facilitate the autonomous learning of divergent optimality, 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 from SU are all relative: there is no available baseline for comparison 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. This implies that the emergent system must actively *explore* the available set of compositions during operation if it is to gain

a complete understanding of the available search space. This online exploration to gain new knowledge must of course be appropriately balanced with exploitation of existing knowledge in order to maintain good overall performance.

Additionally, what is known to be 'good' under one set of external stimuli may be entirely different under other sets of external stimuli. These stimuli may therefore need to be characterised as and when they occur, so that moving performance baselines can be kept for each case, avoiding constant re-learning whenever a change occurs.

3.1.3. Abstracting the environment. 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 found. Online feature extraction is a difficult problem in machine learning [Glocer et al. 2005], with the further problem 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. Further, the way in which environment ranges are detected and distinguished from one another should strive to avoid the classic control loop problem of oscillation between two nearby choices which can expend unnecessary adaptation effort.

3.1.4. Data quality and perception errors. Because we assume that the software emits metrics that describe its health, and events that describe its operating environment, it is possible that the data reported by the software for these purposes is sub-optimal for effective machine learning. As an example, it may be the case that the software finds its performance significantly degrading when nothing about the system's external stimuli has apparently changed, a situation that can occur if the kinds of events reported by u do not measure the kinds of changes that have occurred in the software's operating environment to cause this. These are *perception errors* indicating blind spots in what the machine learning system can observe relative to what it is trying to understand; ideally it should be possible for the machine learning system to detect when it experiences likely perception errors so that adjustments can be made to the kinds of events or metrics that it sees from the software.

3.1.5. Online and offline learning. The key requirement of learning for emergent software is that the system must learn according to what it actually experiences in its normal environment. 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 reassembled 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 load-

ing 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.

3.1.6. 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 one avenue being to selectively 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. This has been explored to some extent in [Porter et al. 2016] but further work is needed to find fully generalised solutions.

3.1.7. Self-referential fitness landscapes. As reported by Cakar et al. in the context of parametric self-optimisation [Cakar et al. 2011], it is likely that changes to the currently chosen composition of u that make up a running system can impact 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).

3.1.8. 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).

3.1.9. Unexpected properties. As reported by Fisch et al. [Fisch et al. 2010], 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 the 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 for the target system under the conditions that it has experienced.

3.1.10. 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. We specifically use *Dana* [Porter 2014] as our runtime component platform due to its affinity for fine-grained components and very fast runtime adaptation. Other component models offering runtime adaptation could also be used, however.

Our emergent software framework is 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.

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. We assume that 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.

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.

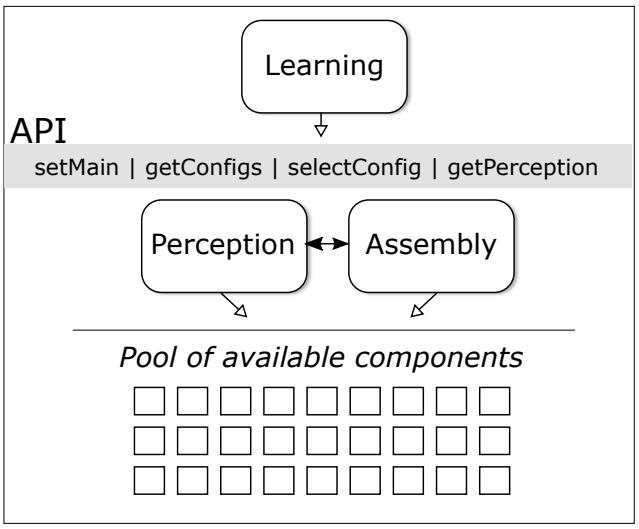

Fig. 1. Architecture of the emergent software framework.

Assembly module This component is responsible for discovering and assembling the target system's components to present a set of candidate configurations, each of which functionally meets the goal of the system. There are various different ways in which an assembly module can perform this discovery of candidate configurations, depending on how the library of available components is organised. In our current implementation, the assembly module 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). During this process, recursive dependencies must be identified as they would otherwise lead to an infinite set of candidate configurations; in our current implementation we stop our discovery process for any branch at the first point at which we detect recursion.

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 inter-connections. 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.

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

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

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

Fig. 2. The Event and Metric data types and the Recorder interface, using the syntax notation of the component model that we use [Porter 2014].

Perception module This component perceives the behaviour and performance of the currently assembled configuration of components, and also the characteristics 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.

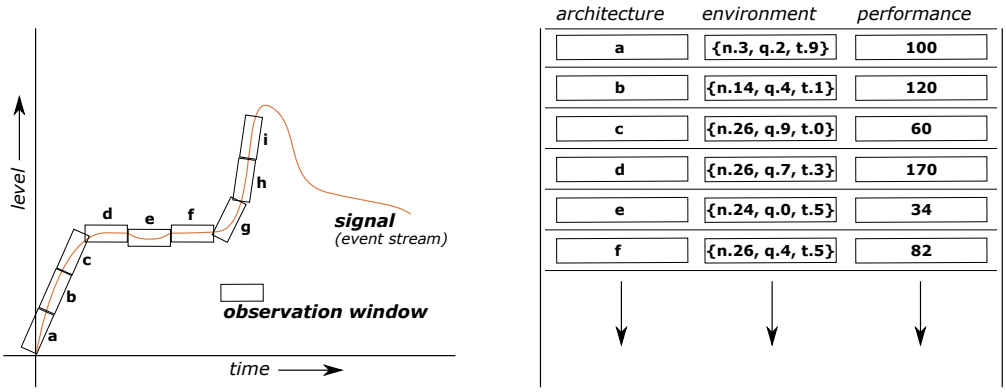

Fig. 3. The emergent software online learning problem. On the left we illustrate the way in which the environment changes over time, and the quantisation occurs over this signal due to the use of learning observation windows. On the right we illustrate the data collected at the end of each observation window, including the environment description and the performance level of the configuration used during that time.

As identified in Sec. 3.1.5, there is a range of possible approaches to learning in emergent software – any of which could be used by the learning module, including hybrid strategies. Here we focus on a purely *online* learning approach in which all learning must be done based on observations and experimentation with the live system. The general form on online learning is to select a configuration for use, using *selectConfig()*, wait a period of time, and then collect perception data to interpret how well the configuration performed. Within this category of learning we can use two strategies, based on observed perception data: reactive and/or predictive. To help understand these strategies we illustrate the online learning problem in its abstract form in Fig. 3. On the left of this figure is the environment (event stream) to which the system is subjected, which is generally outside the control of our system. We make periodic observations over time, resulting in a quantised view of the environment. On the right of Fig. 3 we then show the data collected via the perception module at each periodic observation. In our example here we have assumed here that the configuration being used changes in each successive observation window, either due to exploration or to the exploitation of existing knowledge. From this example a number of challenges are clear:

— **Comparison difficulty:** It is difficult to make comparisons across different configurations in different observation windows as the environment is likely to change over time. During the use of configuration (a), for example, the environment is in a different state compared to the use of configuration (b), making it hard to compare the performance levels of (a) and (b) to decide which is best in each environment range.
— **Mid-window changes:** The environment may change *during* an observation window, as is the case when using architecture (g) where the environment transitions from a flat signal to an upward slope. Depending on the level of perception detail from the system, in the events that describe the environment, this transition may or may not be visible, causing this environment change to be seen as a simple average.
— **Self-referentiality:** While the environment state is generally outside our control, it is still possible that our choice of configuration can have an apparent impact. This is exemplified in the use of configuration (e) which coincides with an apparent dip in environment signal. If this signal is the number of requests seen by the system, the dip could actually be caused by configuration (e) having poorer performance and so a lower throughput in this environment range. The true environment signal may therefore have frequent perturbations caused by the use of different configurations.

— **Observing the past:** It is clear that an emergent system is only able to observe *what has recently happened* rather than knowing *what is currently happening* in the environment. A purely reactive online learning system, which chooses architectures based on what it has most recently seen, thus assumes that the environment will generally continue to behave as it recently did, at least for a short time. A predictive strategy, by contrast, may consider the general upwards or downwards trends and be able to infer whether or not these trends will continue, thereby making decisions on where the environment is potentially going rather than where it has recently been.
— **Hidden trends:** The environment signal shown in Fig. 3 shows clear trends. In reality, however, the kinds of events that the software is reports may not show the trends that are actually key to the way in which the system behaves. Instead the signal of interest may require processing of the primary signal to reveal the way in which the perceived environment really maps onto the perceived performance of the system. An example of this is events that report how many of each kind of request have been seen, where the signal of interest is actually the variation level of these requests.
— **Multi-dimensionality:** Finally, we note that Fig. 3 shows a single environment dimension, as would be reported by a single type of event from the perception module. In reality, event streams tend to be multi-dimensional, reporting multiple different kinds of events that each describe an aspect of the environment, therefore making all of the above challenges themselves multi-dimensional in nature.

As a useful baseline, in this paper we use a simple reactive online learning approach which starts with no prior information about the system. Using this approach our 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 in 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). And second, as in any online learning system, the learning module must balance the tradeoff between exploring options about which there is insufficient information and exploiting options known to be good [Sutton and Barto 1998]. Because our emergent software framework operates on live software, this balance is particularly important because operating sub-optimally has real consequences.

As discussed above, performing both tasks online is highly challenging: the software is not in control of its 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 [Sutton and Barto 1998], 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.

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 $w_t * length(getConfigs())$.

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
```

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, 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 these details. The main problem with our approach comes when the environment changes significantly *during* an exploration activity, meaning that the configurations used were not really compared in the same conditions. This is addressed by our second two uncertainty clauses, 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 [Zheng et al. 2011]. 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.

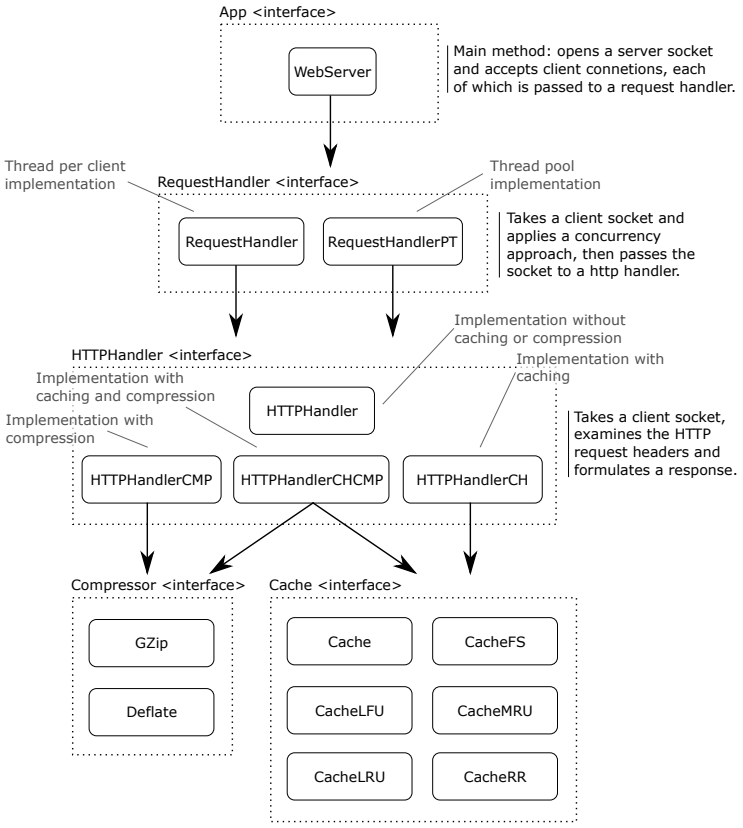

Fig. 4. The set of components from which our web server can emerge. Boxes with dotted lines are interfaces, and those with solid lines are components implementing an interface. Arrows show required interfaces of particular components. The general purpose of each interface's implementations is noted by the interface, and a description of how the available implementation variations of that interface work is also indicated.

Architecture The main components from which our web server can emerge are shown in Fig. 4, 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 to the system and we have two different variants: one that creates a new thread for every client, and one that uses a pool of threads on which to enqueue clients. From the request handler, a client socket is then 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.

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. 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 [NASA 1995]. Our evaluation covers four major aspects of the design space for machine learning.

First, we present a detailed analysis of the ground truth for our system, showing 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. We also show that this phenomena exists not just under different workloads on a single platform, but is also observable as a result of different hardware platforms, where elements such as the relative speed of secondary storage access affect the corresponding performance levels of different compositions of behaviour under varying workloads.

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.

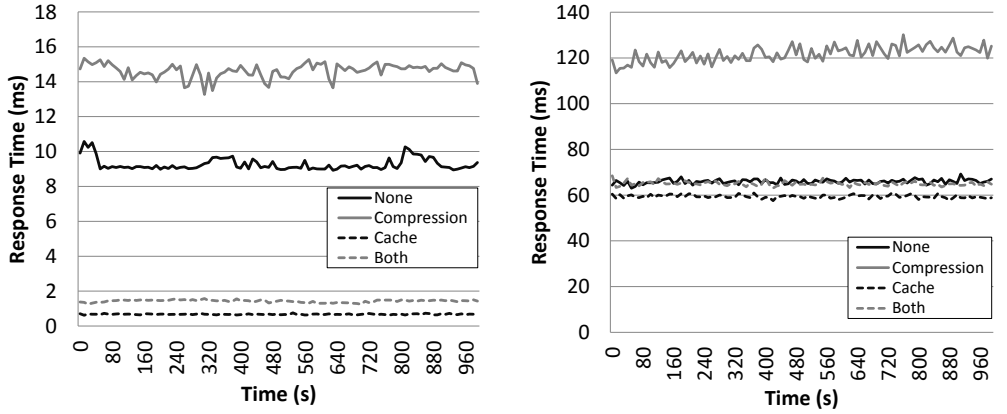

Fig. 5. This graph illustrates the performance of four different configurations with the request pattern of small text files when running on a high-performance server (left) and a Raspberry Pi (right).

Third, we present a deeper experimental evaluation of the issues involved in performing real-time environment classification in combination with a reinforcement machine learning approach. We evaluate our existing approach to classification, using a set of additional tests designed to stress the classifier, and from this we draw lessons for future design directions in this area.

Finally, we examine the effects of observation window sizes on real-time machine learning for emergent software. Here we configure the observation window of our learning algorithm to different sizes and observe the results on convergence time and relative stability of exploitation periods. Again we draw key lessons from this for design directions in real-time machine learning algorithms for emergent software.

All code used in our evaluation, along with instructions on how to repeat all of our experiments, is available at [Filho 2016].

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. 5–7; while results from the NASA trace are shown in Fig. 8. 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. 5 and Fig. 6 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. 5 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. 6 shows that the opposite of this is true. Compared to the results from a Raspberry Pi, shown on the left side of the same figures, we see similar trends but with different separations between each configuration, indicating that the hardware differences are reflected in the relative performance of each configuration. As an example, in Fig. 5 we see a significant separation between the cache+compression configuration and the default configuration when running on a rackmount server platform, while there is almost no separation on the Raspberry Pi platform.

In Fig. 7 and Fig. 8 we show the average response time of the web server for request patterns in which many different files are requested, again illustrating the results on both a rackmount server and Raspberry Pi platform. In detail, Fig. 7 shows results from a custom request pattern in which each request is for a different small ($\sim$3KB) text file; while Fig. 8 shows results from replaying the NASA trace (which also has a

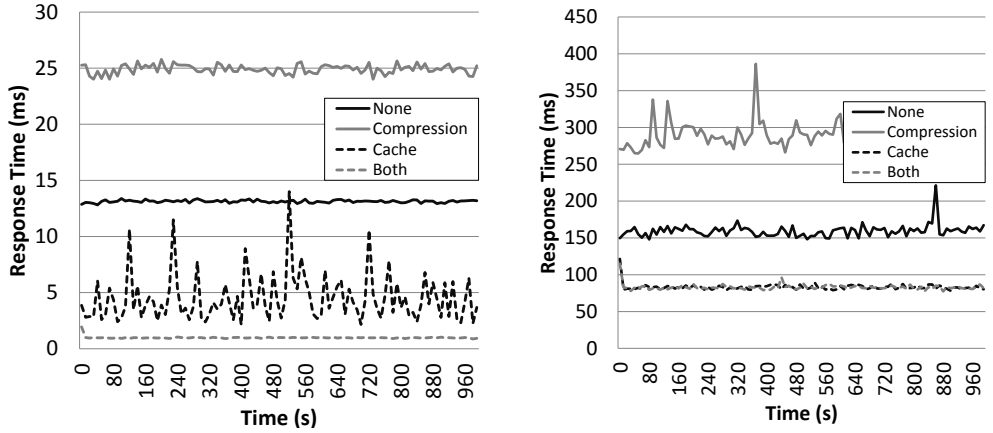

Fig. 6. This graph illustrates the performance of four different configurations with the request pattern of small image files when running on a high-performance server (left) and a Raspberry Pi (right).

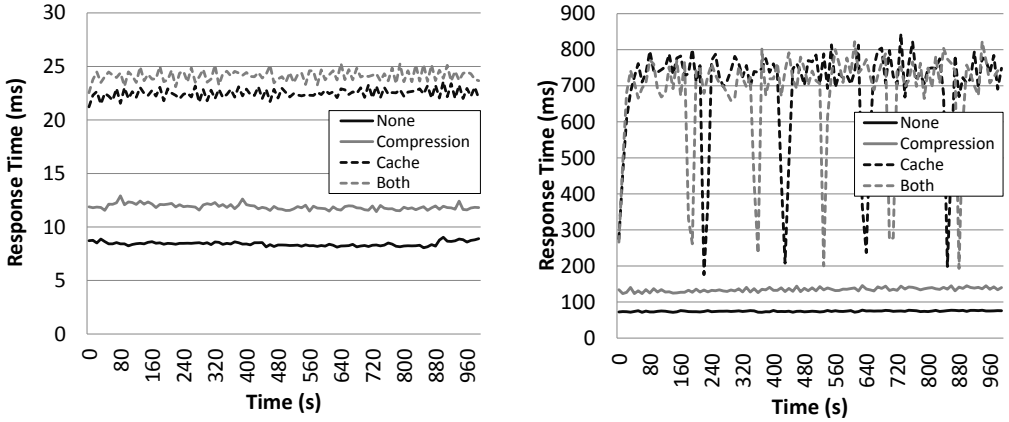

Fig. 7. This graph illustrates the performance of four different configurations with the request pattern of a variation of small text files when running on a high-performance server (left) and a Raspberry Pi (right).

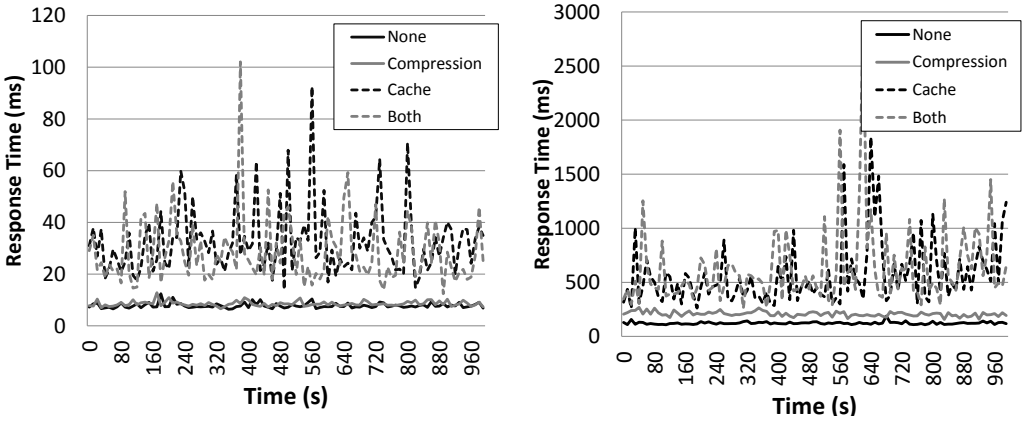

Fig. 8. This graph illustrates the performance of four different configurations with the request pattern of the NASA trace [NASA 1995] when running on a high-performance server (left) and a Raspberry Pi (right).

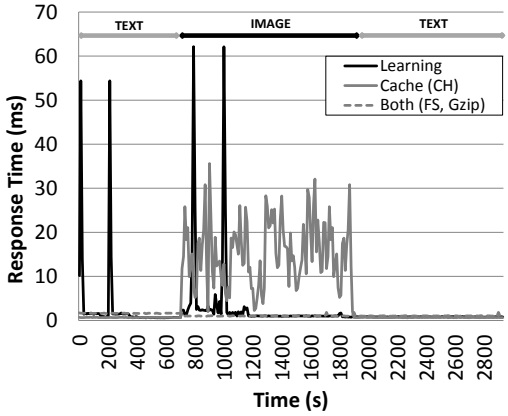

Fig. 9. Performance comparison between fixed web server architectures and our emergent platform, using two different request patterns over time.

high degree of variation). In both of these graphs we see that the best configurations from Fig. 5 and Fig. 6 are in fact the worst two configurations for these request patterns. Both the server and Raspberry Pi results reveal the same general trend, but again the hardware differences put the various web server configurations at vastly different performance points relative to one another.

This clearly demonstrates that different configurations of our web server will perform differently when subjected to different request patterns at runtime. In particular, request 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. Furthermore, this environment includes the physical hardware platform on which the system is running as demonstrated by our Raspberry Pi experiments, a factor which is automatically accounted for by emergent software systems in continually forming the most ideal composition of that software over time.

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. 9 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 infor-

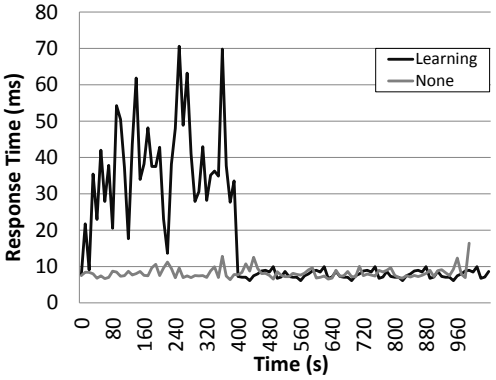

Fig. 10. Performance comparison between a fixed web server architecture and our self-adaptive platform when using the NASA request pattern [NASA 1995].

mation 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. 10 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. 9, 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.

4.3. Real-time classification of environment

In this section we examine our range-based classification approach in more detail. As described in Sec. 3.2.1, we took this approach to mitigate against self-referential fitness landscape issues as well as mid-exploration environment changes.

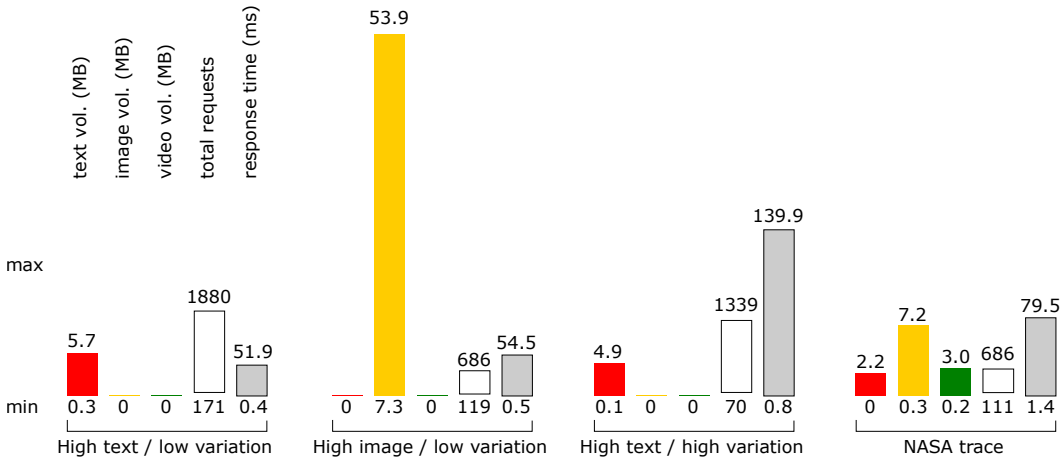

Fig. 11. Classification results: environment ranges detected for each workload.

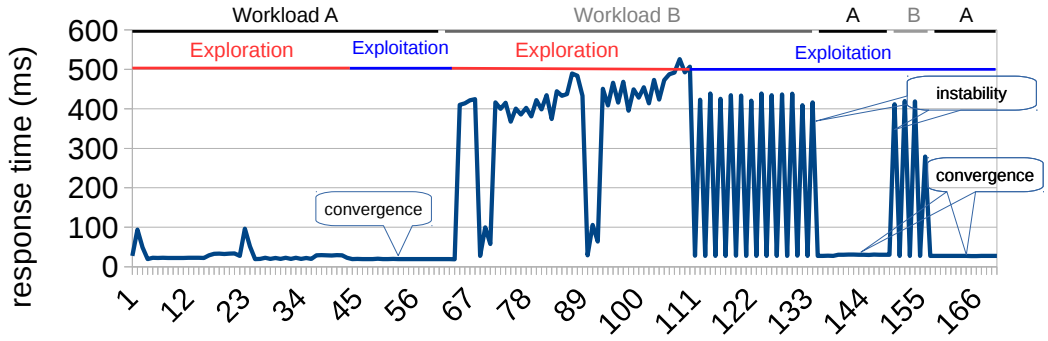

Fig. 12. Classification experiment using request patterns that are difficult to distinguish.

We first examine the classified ranges that were identified for the workloads described in Sec. 4.1. These ranges are shown in Fig. 11, and demonstrate that our classifier is able to detect most of the differences between our example workloads, but not all. Specifically, the low-variation text, low-variation image, and NASA workloads all have different classified environment ranges, allowing their corresponding optimal architectures to be easily distinguished. By contrast, the low-variation text and high-variation text workloads have very similar classified ranges, lacking a clear distinguishing feature; this is because the events that are generated by the web server do not capture the idea of variation (the ratio of adjacent files in a request sequence that are different) within a single media type. Developing ways to infer this kind of attribute from the primary event stream, or otherwise report that additional event detail would be useful to distinguish ranges, are interesting directions for future work.

Next we examine our learning system under difficult environment classification conditions to further highlight the challenges involved. We do this in two different ways. In Fig. 12 we show the possible effects of being unable to distinguish between two environment ranges, as is the case with high- and low-variation text scenarios. At the beginning of this experiment we subject our system to a low-variation text workload (Workload A) and allow the learning system to converge on an optimal configuration

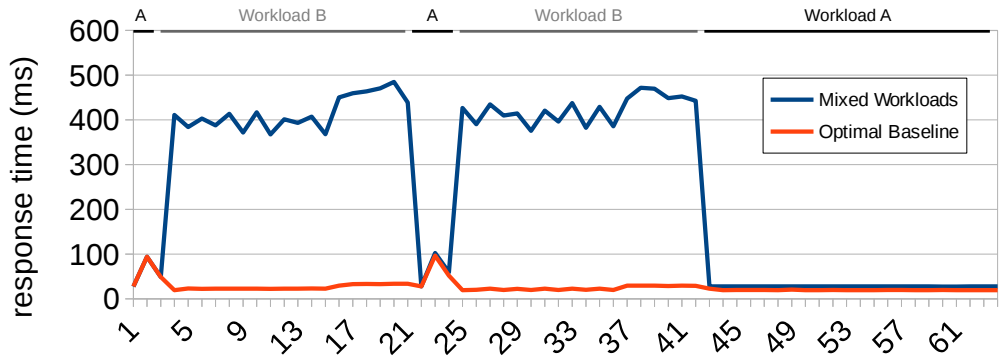

Fig. 13. Classification experiment using mid-exploration workload changes.

(a configuration using caching, in this case). We then change the workload to high-variation text (Workload B). At this point (time 67) the learning algorithm observes a degradation in performance, indicated by the large spike in response time, and waits for three consecutive observations in this performance change. The learning algorithm then triggers a new exploration activity, which finishes learning at time 111 and converges on an optimal configuration for this workload (a configuration that does not use caching, in this case). We then see an undesirable effect: the system oscillates between two configurations, one of which performs very poorly and the other which performs very well. This is because, having chosen the non-caching configuration, this brings response time back down to a low level, within the limits of the response time used in the classified range of Workload A. This triggers the system to assume that the environment is now Workload A, and so selects the best known configuration for that environment, which is a configuration that uses caching. However, this configuration performs very badly under the high variation workload, which in turn pushes response time back into the range constructed for Workload B. This causes the system to assume that the environment is now Workload A, and so selects the configuration that does not use caching. This cycle then repeats until the workload changes at time 140.

The inability to appropriately distinguish environment ranges from one another, combined with the use of response times to help trigger current environment range checks, in this case causes a continuous cycle of poor decisions that need to be constantly corrected. This demonstrates the critical nature of environment classification in emergent software, and the challenge in doing this when the environment's effects are partially coupled with the performance characteristics of the system itself.

In Fig. 13 we show the possible effects of the workload changing during an exploration activity. Here we start the system in Workload A, then during the exploration activity we move to Workload B, then back to A and finally B again. Once the exploration activity finishes we transition back to Workload A. When exploration finishes, we see that the system converges on a configuration that is not quite as good as the known optimal configuration for Workload A, having been unable to accurately compare all architectures under the same conditions during exploration. This highlights the need to better distinguish environment ranges during exploration activities; we also note, however, that the end result of this experiment does still leave the system relatively close to the optimal solution, despite our simple approach to classification, suggesting that this is not as significant an issue as that highlighted in Fig. 12.

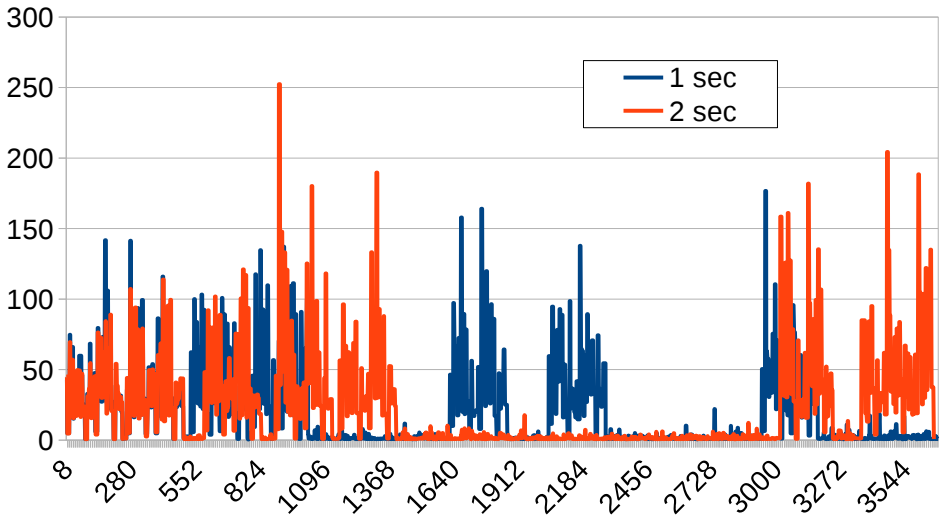

Fig. 14. Observation window experiments of 1 and 2 seconds.

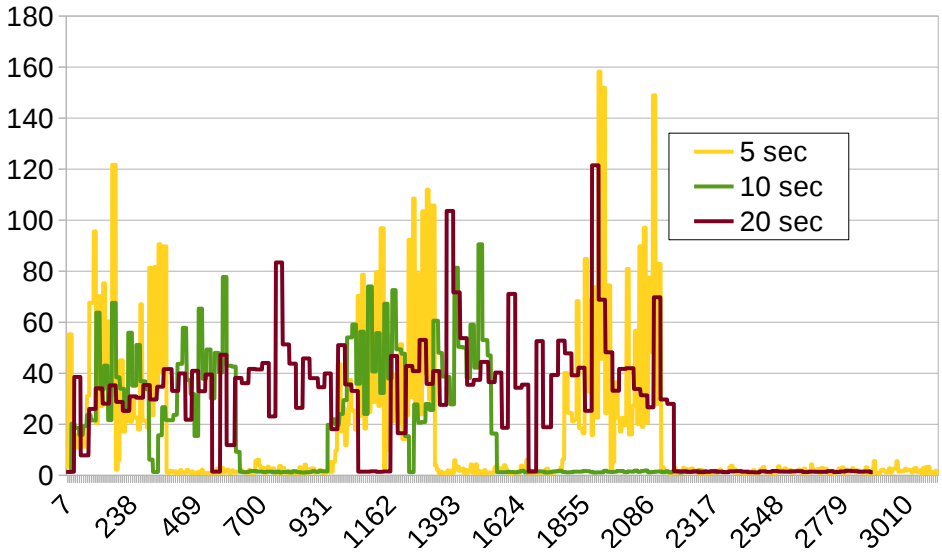

Fig. 15. Observation window experiments of 5, 10 and 20 seconds.

4.4. Window size effects in online learning

In all of the above results we use a common observation window size of 10 seconds. In this section we investigate the use of alternative window sizes to see their affect on learning behaviour. To do this we run our system against the NASA trace using five different observation window sizes, from 1 second up to 20 seconds. The effects of this are shown in Fig. 14 and Fig. 15, with further extracted details shown in Fig. 16. In each experiment we run our system for the same portion of the NASA trace and observe the resulting response time behaviour of the system, the total number of adaptations that are carried out over time, and the number of exploration activities that are triggered.

	1sec	2sec	5sec	10sec	20sec
# of classes	7	6	3	2	2
# of adaptations	480 (186)	376 (124)	141 (15)	104 (20)	84 (0)
Experiment duration	1 hour	1 hour	50 min	39 min	48 min

Fig. 16. Adaptation details of observation window experiments, showing total number of adaptations performed, and adaptations performed under exploitation shown in brackets.

The picture here is relatively complex, with a set of interacting features. Overall we see that the experiment using a 10 second observation window size completes the trace fastest, indicating that it had the best overall response times. There are two different reasons for this. First, the use of smaller observation windows causes *over-reaction* to the variations in the workload, and an unnecessarily large number of adaptations. These adaptations, while cheap, do momentarily impact the response time of the web server, resulting for example in the 5 second experiment having far higher peaks than the 10 second experiment shown in Fig. 15. The data for the 1 and 2 second window sizes, in Fig. 14, show even more volatility as the system attempts to over-fit to the environment. Second, the use of larger observation windows causes *under-reaction* to the variations in the workload. This is exemplified by the 20 second observation window experiment, which has two exploration phases, but never performs any adaptations outside of those phases during exploitation, indicating that it has classified environment ranges that are very broad and so miss important details.

This demonstrates that the configuration of the optimal observation window size is itself a challenge, and may have a different ideal setting across different types of application, different workload characteristics, and different hardware platforms. Beyond the static observation windows used here, it may also be useful to explore dynamic observation window sizes, which continually balance both over- and under-reaction.

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. 3.1.5) that takes into account the self-referential fitness landscape issue (Sec. 3.1.7).

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.

ACKNOWLEDGMENTS

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

Nelly Bencomo and Amel Belaggoun. 2014. A World Full of Surprises: Bayesian Theory of Surprise to Quantify Degrees of Uncertainty. In *Proc. of the 36th International Conference on Software Engineering (ICSE Companion 2014)*. ACM, New York, NY, USA, 460–463.

Nelly Bencomo, Amel Belaggoun, and Valerie Issarny. 2013. 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*. 113–122.

Emre Cakar, Sven Tomforde, and Christian Müller-Schloer. 2011. A role-based imitation algorithm for the optimisation in dynamic fitness landscapes. In *IEEE Symposium on Swarm Intelligence (SIS)*. 1–8.

Bihuan Chen, Xin Peng, Yijun Yu, Bashar Nuseibeh, and Wenyun Zhao. 2014. Self-adaptation Through Incremental Generative Model Transformations at Runtime. In *Proc. of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 676–687.

Ahmed Elkhodary, Naeem Esfahani, and Sam Malek. 2010. 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*. ACM, New York, NY, USA, 7–16.

John M. Ewing and Daniel A. Menascé. 2014. 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 (ICPE '14)*. ACM, New York, NY, USA, 173–184.

Funmilade Faniyi, Peter R. Lewis, Rami Bahsoon, and Xin Yao. 2014. Architecting Self-Aware Software Systems. In *Proceedings of the IEEE/IFIP Conference on Software Architecture (WICSA)*. 91–94.

Jacques Ferber. 1999. *Multi-agent systems: an introduction to distributed artificial intelligence*. Vol. 1. Addison-Wesley Reading.

Roberto Rodrigues Filho. 2016. Source code from this paper with instructions: http://research.projectdana.com/taas2016rodrigues. (2016).

Dominik Fisch, Martin Janicke, Bernhard Sick, and Christian Muller-Schloer. 2010. Quantitative Emergence – A Refined Approach Based on Divergence Measures. In *Proceedings of the 4th IEEE International Conference on Self-Adaptive and Self-Organizing Systems*. 94–103.

Karen Glocer, Damian Eads, and James Theiler. 2005. Online Feature Selection for Pixel Classification. In *Proceedings of the 22nd International Conference on Machine Learning (ICML '05)*. ACM, New York, NY, USA, 249–256.

Paul Grace, Danny Hughes, Barry Porter, Gordon Blair, Geoff Coulson, and Francois Taiani. 2008. Experiences with open overlays: a middleware approach to network heterogeneity. In *Proc. of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*. 123–136.

Sara Hassan, Nelly Bencomo, and Rami Bahsoon. 2015. 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*. 134–145.

Mike Hinchey, Sooyong Park, and Klaus Schmid. 2012. Building Dynamic Software Product Lines. *IEEE Computer* 45, 10 (Oct 2012), 22–26.

Olga Kouchnarenko and Jean-Francois Weber. 2014. Adapting component-based systems at runtime via policies with temporal patterns. In *Formal Aspects of Component Software*. Springer, 234–253.

D. Menasce, H. Gomaa, S. Malek, and J.P. Sousa. 2011. SASSY: A Framework for Self-Architecting Service-Oriented Systems. *IEEE Software* 28, 6 (Nov 2011), 78–85.

NASA. 1995. NASA web server trace: http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html. (1995).

Barry Porter. 2014. Runtime Modularity in Complex Structures: A Component Model for Fine Grained Runtime Adaptation. In *Component-Based Software Engineering*. ACM, 26–32.

Barry Porter, Matthew Grieves, Roberto Rodrigues Filho, and David Leslie. 2016. RE$^{\text{X}}$: A Development Platform and Online Learning Approach for Runtime Emergent Software Systems. In *Symposium on Operating Systems Design and Implementation*. USENIX, 333–348.

Barry Porter and Roberto Rodrigues Filho. 2016. Losing Control: The Case for Emergent Software Systems using Autonomous Assembly, Perception and Learning. In *International Conference on Self-Adaptive and Self-Organizing Systems*. IEEE, 40–49.

Mazeiar Salehie and Ladan Tahvildari. 2009. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 4, 2 (2009), 14.

Richard S. Sutton and Andrew G. Barto. 1998. *Reinforcement Learning: An Introduction*. Bradford Book.

Sven Tomforde, Jörg Hähner, and Christian Müller-Schloer. 2013. 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*. 185–192.

Yiqiao Wang and John Mylopoulos. 2009. Self-repair through reconfiguration: A requirements engineering approach. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 257–268.

Wei Zheng, Ricardo Bianchini, and Thu D Nguyen. 2011. MassConf: automatic configuration tuning by leveraging user community information. In *ACM SIGSOFT Software Engineering Notes*, Vol. 36. ACM, 283–288.