A Runtime Framework for Machine-Augmented Software Design using Unsupervised Self-Learning

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

I. Introduction

Modern computer software comprises tens of millions of lines of code, created by large development teams in major multi-year projects. This represents a level of *complexity* that is moving beyond the reach of humans to understand.

This growing complexity in software systems has motivated the development of autonomic computing concepts and practice. The vision of autonomic computing aims to shift the burden away from humans and into software itself for tasks including software installation, maintenance, configuration and management [1]. A large body of work has now been done around 'self-*' properties towards this vision [2]. However, existing work either (i) does not consider the design process, targeting a specific problem in isolation such as parameter tuning [3], therefore neglecting the broader complexity in software construction; or (ii) considers the design process but is human-centric, relying on architectural or feature models that retain significant human involvement [4]. While frameworks have been proposed to infuse software with increasing levels of autonomy [5], our focus is on having autonomous processes take a leading role in the task of software design itself.

We argue that a more comprehensive and *machine-centric* approach is needed to take the next step towards automation in software design, development and maintenance. Our work therefore examines software development as a process, infusing this process with a level of autonomy that seeks **to make software an active member of its own development team**.

We present an overview of our framework, highlighting how autonomous computing can support different stages of software development, and we demonstrate the accuracy of our framework in autonomously finding the most suitable software design at runtime according to specific operating conditions.

II. Framework

Our framework considers the major activities in conventional development: specifying **requirements**; realising those requirements through **implementation**; and managing/maintaining the system in its **deployment** environment. In each stage it is designed to interact with human engineers, but maintains a leading role in the design directions taken.

1) Requirement stage: Requirements are viewed as an abstract intent for the system alongside any constraints over how that intent is realised, supplied by a developer or end user. This stage acts as the human interface to the development process: translating between human language and conceptualisations and machine-driven decision making and communication. Following appropriate translation, the intent and constraints are passed to the implementation and deployment stage, described shortly. Live feedback to the requirements stage may arrive from the deployment stage, providing high-level summaries of autonomous design choices that are being made in order to ascertain whether or not these choices remain in the spirit of the specified intent and constraints – or potentially indicating if some constraints are not workable in the deployment.

2) Implementation stage: This stage takes the output of the requirements stage and generates a working system for deployment. Our system assembly methodology uses component-based design to produce many small pieces of behaviour from which to assemble a system. Our current implementation relies on humans to do this, producing enough components to assemble an initial working system. The implementation stage then receives feedback from the deployment stage in the form of new component *variation requests*. These requests indicate a desirable design point exploration that has been identified in the running system, complete with the characteristics of the deployment environment for which this variation is requested to perform well. The implementation stage then attempts to generate a component matching these requirements, or else report that it is impossible to create a component that performs at the specified level in the given conditions – in which case the deployment stage will consider other options.

3) Deployment stage: The deployment stage is responsible for assembling the application architecture, monitoring its behaviour, exploring the different available design choices in the operating environment, and learning about the application behaviour in each detected environment. The deployment stage leads the design process from actual experience, reporting its findings higher levels of the framework (including humans). The deployment stage is our most complete element, comprising an assembly module, a perception module, and a learning module, plus a knowledge base and a design variant analyser.

The **assembly module** is responsible for building the target software system and re-assembling it with alternative designs when applicable. The **perception module** observes and records the characteristics of the deployment environment and performance of the software. The **learning module** then

discovers which assemblies work well in which deployment environment characteristics based on the monitored data, with pluggable machine-learning strategies. The **knowledge base** stores analysis results of from learning module, including design rules derived for different environments. Finally, the **design variant analyser** examines data held in the knowledge base and identifies likely high-reward points for the software which would benefit from component or design variations. Note that all of these activities are designed to operate on a live software system deployed in its production environment.

III. Case study and Evaluation

As a case study of machine-augmented software design, we have applied the deployment stage of our framework to a web server built with a runtime component model [6] such that many of its constituent components may have variations – different implementations of the same interface which likely have different performance characteristics in different deployment conditions. Our web server has over 40 different possible architectural designs, including variations in components that implement cache replacement strategies (e.g. least frequently used) and compression algorithms (e.g. gzip). The deployment stage of our framework dynamically discovers all possible components from which to assemble the system and, with no prior application-specific knowledge, experiments with the different assemblies whilst analysing the incoming requests. This autonomous analysis results in dynamically created adaptation rules, correlating the appropriate assembly variant to each detected range of operating conditions.

Our initial evaluation is conducted by exposing all available architectural designs to a real-life web traffic trace, for which we used the NASA trace available at [7]. We configure a client application which makes sequential requests for the resources listed in the trace. The results shown in Fig. 1 present example performance divergence of four groups of architectural configurations for part of this trace: one group that uses compression, one that uses caching, one that uses both caching and compression, and finally, one group that uses neither caching nor compression. Note that other parts of the trace may have these divergent optimals reversed, such that (for example) architectures with caching or with both caching and compression perform better than others.

The following graph, in Fig. 2, depicts web server performance under the control of our deployment stage, which autonomously searches for the optimal design. A baseline is shown for comparison, indicating the performance of the (manually determined) best fixed configuration of our web server for this part of the NASA trace. The graph shows our deployment stage autonomously experimenting with various possible assemblies of the web server, drawn from different combinations of available components, before converging on an assembly with similar performance to the baseline.

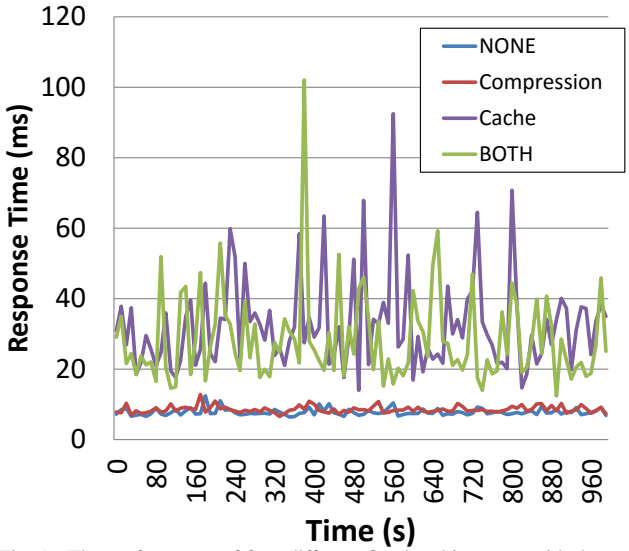

Fig. 1. The performance of four different fixed architectures with the request pattern from part of the NASA trace [7].

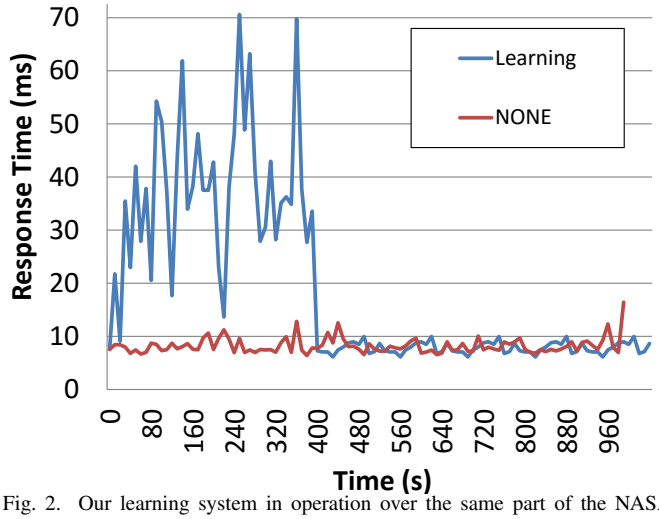

Fig. 2. Our learning system in operation over the same part of the NASA trace [7], with an optimal baseline configuration for comparison.

Acknowledgment

Roberto Rodrigues Filho would like to thank his sponsor, CAPES, Brazil, for the scholarship grant BEX 13292/13-7.

This work was partially supported by the UK's EPSRC under research grant number EP/M029603/1.

References

[1] A. G. Ganek and T. A. Corbi, "The dawning of the autonomic computing era," *IBM Syst. J.*, vol. 42, no. 1, pp. 5–18, Jan. 2003.

[2] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, pp. 14:1–14:42, May 2009.

[3] L. Li and L. Gruenwald, "Self-managing online partitioner for databases (smopd): A vertical database partitioning system with a fully automatic online approach," in *Proceedings of the 17th International Database Engineering & Applications Symposium*, ser. IDEAS '13. ACM, 2013, pp. 168–173.

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

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

[6] 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.

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