ACT: a Tool for Performance Driven Evolution of Distributed Applications (Position Paper)

Aled Sage, Graham Kirby and Ron Morrison
School of Computer Science, University of St Andrews, North Haugh, St Andrews KY16 9SS, Scotland
email: {aled,graham,ron}@dcs.st-and.ac.uk

Abstract

There are two main stages to evolving distributed applications in the manner desired by application builders: first deciding which changes are required and when, and second making the changes. Understanding the performance characteristics of distributed applications is essential for the first stage, while structural reflection over the source code may be used to achieve the latter. Here we present an automated configuring tool, ACT, that may be used to explore the need for change by empirically measuring application performance. We aim to use the data generated by ACT as input to the evolution process, informing the system how to evolve to new and improved architectural configurations.

ACT is designed to be generic in that it may aid performance-driven evolution for a wide range of applications. As a case study we use DC-Mailbox, a back-end mail server from Data Connection Limited (DCL) that stores, retrieves and manages e-mail messages for a potentially large number of users.

1. Introduction

Understanding the performance of distributed applications is essential for achieving the evolutionary properties desired by application builders. The choice of platforms, network configurations and communication policies has increased markedly over the years leading to a wide variety of operating environments with ever more complex performance characteristics. This makes the problem of performance-driven evolution more complex and time consuming and therefore more costly, often requiring advanced knowledge of the application.

Here we present a tool for measuring the performance (in terms of quality of service criteria) of distributed applications. We aim to use the data generated by the Automated Configuring Tool, ACT, as input to the evolution process, informing the system how to improve the quality of service (QoS) by evolving to new and improved architectural configurations.

There are varying levels and complexities of evolution. The simplest is to reconfigure the application using pre-defined tuning knobs, or *configurable parameters*. This requires understanding of the application's behaviour, for example by empirical measurement of performance, in order to make beneficial changes. When good configurations are found, the application may be evolved (by some process) so that it can dynamically adapt in a given situation. As understanding is gained, unexpected or interesting behaviour such as phase changes may be investigated and exploited to suggest improvements to the structure of the application.

We are concerned with the task of configuring and evolving applications, the *target systems*, to perform well in customers' environments. Each customer may use different platforms and workloads requiring different configurations of the target system for optimal performance. Examples of configurable parameters explored to date include cache size, tracing policies and number of processors. The problem space in a given environment therefore consists of the configurable parameters, forming the multi-dimensional *input space* of configurations. Choosing a good configuration (that is, one that satisfies the evolution goals such as optimising performance) out of the large number of possibilities is a non-trivial task; empirical testing of different configurations to search this space will help. Good configurations may be used to drive the system's evolution including embedding adaptive control into the evolved application.

The tool, ACT, aims to ease the problem of running performance tests, or *trials*, many times in a consistent manner. ACT does this before the target system goes into use by repeatedly running it, recording performance for specified workload(s) and evolving the system by reconfiguration between trials. Initially, we concentrate on applications that already have performance metrics built-in, or for which it is realistic in terms of time and effort to add suitable measuring facilities. Eventually we aim to use the results of the measurement stage to drive higher levels of evolution in the system.

ACT is designed to be generic in that it may aid performance-driven evolution for a wide range of applications. This is achieved by restricting ACT's use to all target systems that meet certain requirements. It

requires that hooks, or *plug-ins*, be made available to run, measure and (re)-configure the target system. For automation, the plug-ins must run without human intervention. Details of how to access these hooks, and information about the configurable aspects of the application are encoded in an XML file as input to the tool.

The novel aspect of the work is the range and type of reconfiguration. We are particularly interested in applications with many configurable parameters and therefore a large number of possible configurations. These are difficult to analyse because the number of possible configurations increases exponentially with the number of parameters. One must be careful of interference between parameters, ensuring that they can each be changed individually. For some applications, parameters may require a non-trivial length of time to reconfigure (e.g. requiring reinstallation of the application) and each trial may take a significant amount of time. For example, an application with 5 parameters, each with 10 possible values and requiring 30 minutes per measurement would take over 5 years to test exhaustively. This makes automation important and searching the space a key issue, performing tests in parallel where possible and carefully selecting configurations. ACT may be beneficial when the behaviour of the application is not understood, since explorative changes can be made during analysis. Indeed, it may never be possible to fully understand the behaviour of some complex distributed applications operating in many environments. Lastly, each test-run can be started from the same initial state and under the same conditions allowing all relevant variables to be controlled, the only parameters to change being those explicitly set between trials by ACT.

As a case study we use DC-Mailbox [3] a product from Data Connection Limited (DCL). DC-Mailbox is a back-end mail server that stores, retrieves and manages e-mail messages for a potentially large number of users.

2. Dynamic and static configuration

Related work on performance-driven configuration, or *tuning*, ranges from ad hoc manual techniques using tools for investigating performance bottlenecks to generic automated performance-tuning tools. Configuration can be performed dynamically while the application is in use or statically as is currently the case with ACT.

Dynamic configuration involves on-the-fly steering of an application to cope with new circumstances and to meet new demands. Most use a predictive model of the application's behaviour, examples including [1] and [6]. ACT, because of its experimental approach to finding good configurations, can perform analyses and suggest changes even before a predictive model is developed. This is particularly useful due to the complexity of developing and maintaining such models for large distributed applications with many parameters. One must deal with discrete events (e.g. message arrival) and non-linearities (e.g. optimal cache size as message size fluctuates), while different releases of the application may have radically different performance. The problem is compounded if the cost of taking a measurement or reconfiguring the target system is high.

When analysing configurations statically, care must be taken in choosing workloads to ensure they are representative of the customer's usage. It could be done, for example, by logging the usage of the actual system to generate a set of inputs for the trials, or to guide the production of synthetic workloads.

3. The ACT tool

In a given environment and for a particular workload, ACT may be used to aid performance-driven evolution of a (distributed) application. The application is run repeatedly for different configurations and those giving the best results may be used when evolving the system. The process is automated: running and reconfiguring the application for repeated tests requires no human intervention.

3.1. Tool architecture

The system can be divided into two disjoint sets of concerns. The first is application-specific and the responsibility of the tester. It involves determining how to run the application, how to measure the performance and evaluate the result, which configurable parameters are available and how to change them. The second set of concerns constitute the tool itself and relate to how to repeatedly run performance trials and which configurations to test when searching the input space. It is also concerned with which configurations to recommend and time limits for testing.

For an application to be compatible with ACT it must be configurable; a third party must be able to access hooks to run, measure and change the target system. Some hooks may be reused for different applications and others dynamically generated using structural reflection [5], also called linguistic reflection, over the source code. The hooks are provided in a *plugger*, a dynamically linked library (DLL), each instance of which is specific to a target system and contains functions to perform the above operations. Figure 1 shows the main components required and how they interact.

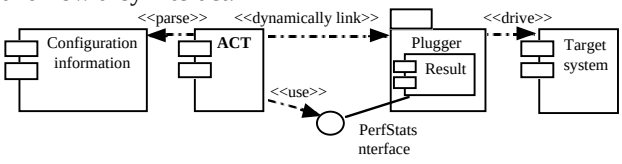

Figure 1. Component diagram of the configuration measurement stage

The *configuration information*, in the form of an XML file written by the tester, describes the legal configurations for a particular application and the information required by ACT to dynamically link in the plugger. For each configurable parameter, a name is given (for meaningful output) along with its type and the set of legal values, specified as an enumeration or a range. To allow dynamic binding of the plugger's application-specific code, the location of each function is given by specifying its name and the path of the DLL. Additional information includes time and machines available during testing.

ACT is responsible for repeatedly running performance trials and searching the input space by selecting new configurations of the application to test. It dynamically loads the plugger and uses it to interact with the application.

The plugger provides the hooks to configure and drive the application. It contains a *run function* to run and gather performance measurements for a single trial. This prepares and starts the application (setting it to a consistent starting state). It then monitors the application's performance to obtain appropriate performance statistics. The measurement technique used depends on the particular application but, where possible, existing techniques are utilised such as in-house tools or profilers. The results are represented by a class that implements the *IPerfStats* interface in figure 1 to provide display and comparison methods, allowing ACT to process the results independently of the application concerned. Additionally, a set of *change functions* is used to set the values of the configurable parameters, thus configuring the application. Finally, a *recovery function* is responsible for recovering from failure during a test by restoring the application to a stable state so that testing can continue.

3.2. Configuring an application

The activity diagram in figure 2 shows the steps in configuring and evolving a target system, here generalised to any configurable process. It has much in common with both process modelling and control systems. Indeed, the process may be viewed as a disturbance-compensated closed-loop control system with command compensation [4], where one or more components may be idle.

During *initialisation*, the configuration information is parsed to determine the configurable aspects of the application and the locations of the plugger's functions, which are then dynamically bound into ACT.

The *control search* activity is a *meta-strategy* that determines an appropriate *search strategy* to look for good configurations in the input space. That is, it is a meta-process responsible for controlling the choice of strategy along with any configurable aspects of the search. The simplest example involves using a statically determined search strategy and simply passing it the configuration and performance information. Alternatively the search strategy may be tuned or dynamically switched according to the problem space, resources available and performance measured to date. Indeed, an instance of ACT itself may be used as a meta-strategy. At present the implementation is limited to statically determining the search strategy, dynamic switching still being under development.

The *get configuration* activity, performed by the active search strategy, generates a configuration to be used in the next trial. It is therefore responsible for choosing from the input space the configurations to test during analysis. The target system is then configured and run. Failure is detected by catching runtime errors and also by a timeout mechanism that puts an upper bound on the length of time per trial, the limit being defined in the configuration information. Additionally, a *disturbance compensator* may be used to monitor the trial and could, for example, observe network or CPU usage to detect when the external load exceeds an acceptable threshold level.

If the run fails, the *recovery function* is invoked to restore the system to a stable state. A maximum number

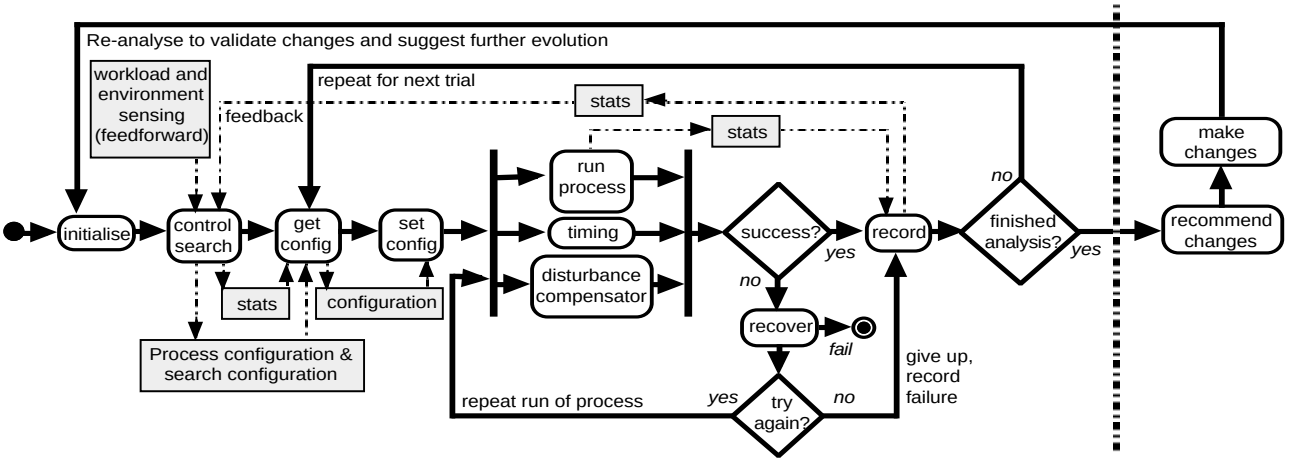

Figure 2. Activity diagram of the configuration and evolution process

of recoveries per configuration are permitted (specified in the configuration information) after which a failure is recorded. If the trial succeeds, the statistics obtained are recorded. The result is fed back to the search controller and the search strategy queried to check if analysis is finished. This cycle repeats until analysis is complete.

The last steps shown in figure 2 involve recommending and making architectural changes to the target system. This high level of evolution has not yet been incorporated, the techniques proposed being described in section 3.4.

3.3. Searching the input space

Analysis involves running trials for different configurations in the multi-dimensional input space in an attempt to find good configurations. This is a search problem in a space that may contain many choices of configuration and with a time constraint limiting the number of points that can be investigated. There are several possible search strategies that may be chosen by the meta-strategy.

The simplest strategy is grid sampling, where the input space is overlaid with a grid and each point on the grid is tested in turn. Even such a basic search may find a configuration that is better than the tester's best guesses and therefore be considered a success. It also provides a control with which to compare other strategies. Alternatives include gradient descent with simulated annealing and genetic algorithms. Both grid sampling and gradient descent are currently being tested.

3.4. Driving evolution

The performance information and good configurations found may be used to drive evolution. The initial implementation of ACT is limited to the simplest level of evolution: reconfiguring specified parameters. This work will lead, as part of the ArchWare project, to the task of recommending architectural changes for the target system. The π-SPACE architecture description language [2] will be used to describe the evolution. Structural reflection [5] over the application source code will be used to perform evolution.

4. A case study

The motivating example for the automated configuring tool is the Data Connection application DC-Mailbox. This back-end mail server is designed to run on a large distributed system and has a broad customer base leading to a wide variety of potential environments in which it must perform well. The experimental base at St Andrews is a 64 node Beowulf running RedHat 7.1 and connected through a fast Ethernet switch. Each node is a Pentium II 450MHz machine with 384MB of RAM and a 6.4GB hard disk.

To be adaptable for different environments, DC-Mailbox exposes a large number of parameters by storing its configuration information in an X500 directory, DC-Directory [3], that is read at startup. Measurements are taken by configuring DC-Mailbox to write to DC-Directory at regular intervals (e.g. every five minutes) the number of deliveries and fetches performed. A synthetic workload is applied by simulating simultaneous access of many users sending and retrieving mail. To test DC-Mailbox in a new customer's environment takes thirty minutes per trial, starting it on each machine, letting performance stabilise and then taking a measurement. Analysis and tuning by hand is therefore time-consuming and relies upon (costly) expertise.

The use of ACT is currently being tested for DC-Mailbox and the above process has already been automated for a selection of configurable parameters. The configuration is set and the performance measured using in-house scripts to read and modify the data stored in DC-Directory. DC-Mailbox itself is started and stopped using a remote shell and a consistent state obtained each time by replacing the database of messages with a clean set of files while also clearing any queues of messages awaiting processing. In the event of recovery, appropriate diagnostics are collected and the application's processes are terminated, ready to be restarted again.

To automatically run, configure and measure DC-Mailbox, the plugger consists of 500 lines of shell scripts and a further 260 lines of C++ code. This is less than one percent of the total application code.

5. Conclusions and further work

In this paper we have described the architecture of a generic automated configuring tool, ACT. The motivation for ACT is to configure an application's performance for a given environment and workload(s), and to use the information obtained to drive evolution. Of particular interest are applications with many configurable parameters installed on a wide variety of platforms, each potentially requiring different configurations to obtain good performance.

At present, ACT's design assumes that the target system has hooks to modify the configurable parameters and that it can be run, measured and the performance evaluated automatically with no human intervention. This has been shown to be an achievable set of criteria for DC-Mailbox. By repeatedly running performance trials and speculatively varying the configuration each time, reconfiguration may be beneficial even before a predictive model of the application is known. We contend that such empirical testing of a number of configurations will aid understanding of the target system for simple reconfiguration and eventually for higher levels of evolution, and hope this will be shown in the results of the case study.

In the mean time, further work is required to test search strategies and visualisation of the results, as well as using the results to drive evolution. We also hope to show the genericity of the tool by its use with other applications currently under investigation, particularly examples with reflective capabilities. More information can be found at [7].

6. Acknowledgements

This work was supported by the EPSRC grant GR/L13780 "Distributed Software Systems" and a grant from Data Connection Limited (DCL). The EPSRC CASE studentship 99802449 also helps fund this project. Useful discussions were held with Richard Stamp, Edward Hibbert, Al Dearle and Francis Vaughan.

References

[1] J.P. Bigus, J.L. Hellerstein, T.S. Jayram and M.S. Squillante, "AutoTune: A Generic Agent for Automated Performance Tuning", *Practical Application of Intelligent Agents and Multi Agent Technology* (2000)

[2] C. Chaudet, R.M. Greenwood, F. Oquendo and B.C. Warboys, "Architecture-driven software engineering: Specifying, generating, and evolving component-based software systems", *IEE Proceedings – Software Engineering, Vol. 147, No. 6* (2000)

[3] DCL. "Data Connection Limited (DCL)". URL: *http://www.dataconnection.com*

[4] E.O. Doebelin, *Control System Principles and Design*, Wiley & Sons, New York (1985)

[5] G.N.C. Kirby, *Reflection and Hyper-Programming in Persistent Programming Systems*, Ph.D. Thesis, *University of St Andrews*, (1992)

[6] D.J. Kerbyson, E. Papaefstathiou and , E. G.R. Nudd, "Application Execution Steering Using On-the-fly Performance Prediction", *Proc. High Performance Computing and Networking 98*, Amsterdam, Holland (1998).

[7] A. Sage, "ACT: An Automated Configuring Tool", URL: *http://www.dcs.st-and.ac.uk/~aled/ACT/index.html*