System Evolution, Feedback, and Compliant Architectures

R. Mark Greenwood, Ken Mayes,
Brian C. Warboys, Ben S. Yeomans
Department of Computer Science
University of Manchester
Oxford Road, Manchester, M13 9PL
+44 161 275 6183
{markg,ken,brian,yeomansb}@cs.man.ac.uk

Dharini Balasubramaniam, Graham Kirby,
Ron Morrison
School of Computer Science
University of St Andrews
North Haugh, St Andrews, Fife, KY16 9SS
+44 1334 463240
{dharini,graham,ron}@dcs.st-and.ac.uk

1 Introduction

The thesis of this position paper is that the evolutionary development of a software system can be significantly affected by the compliance of its software architecture. The notion of compliance focuses on the relationship between a software system being developed or evolved, and system functions, such as concurrency control, scheduling, address space management and recovery management, which are often provided by components, e.g. languages, operating systems and libraries, over which the system developers have little or no control [4]. Where there is compliance the system functions match the requirements; with non-compliance additional code is required to bridge between the system functions provided and those which are required. This will be illustrated through some details of what has happened in the development of the Process*Web* system [5], concentrating on the issue of scheduling.

Process*Web* is a multi-user system which provides process support through the execution of process models [6]. It is based on ICL/TeamWARE's ProcessWise Integrator (PWI) engine extended to allow web browsers to be used as the user interface. Its users can login and logout from the system at will, irrespective of the state of any process in which they are participating. Users can also be involved in many distinct processes and can switch their focus between them at will. To provide this flexibility Process*Web* consists of many threads which need to be scheduled.

The novelty of a compliant architecture is that it is designed top-down to meet the needs of the system. This contrasts with the more traditional approach where components are designed to deliver system functions based on assumed needs. With a compliant architecture it would possible to employ a scheduling algorithm tailored to Process*Web* using the basic mechanisms provided. For example, it might be appropriate to give priority to threads which related to users currently logged into the system. However, currently Process*Web* does not have a compliant architecture. Specifically it is implemented in a multi-threaded language PML, and PWI's PML interpreter imposes a round-robin scheduling policy on all runnable threads.

For the developers this scheduling policy is simply part of the environment which they have to take into account. Where there is feedback which indicates that this policy is not what is required, the developers introduce new code to alleviate the problem. The lack of architectural compliance affects the system's ongoing development. This is significant because often the new code introduced is among the hardest to understand and this causes knock-on effects for other system changes. To illustrate in more detail, we must briefly outline part of the Process*Web* implementation.

2 Process*Web* Architecture Overview

We will concentrate on the user interface aspects of the architecture. As a very visible part of the system, the user interface is particularly prone to user feedback of faults or inconveniences with the system. In addition, delivering an adequate response to users is one of the main aims of scheduling, on which we are concentrating.

Users interact with Process*Web* through web browsers. The user interface for the executing process models is based on HTML which is sent to users' browsers. At the simplest level, there are general facilities which enable users to browse around their personal view of the executing models. This view is in terms of HTML pages, and the browser issues a request which "pulls" any new HTML page required. When a user provides input which changes the process state then a revised HTML page will be generated, both for the user who provided the input and for all other users affected by the state change. In this situation, Process*Web* will "push" the updated pages if these are currently being viewed by any of the other users.

Input is through HTML forms and uses standard CGI (Common Gateway Interface) mechanisms. The user interface mechanisms are the same for all the models and Process*Web* provides a number of standard facilities which hide the communication details from the models.

The executing process models, and Process*Web* itself, are implemented in PWI's PML language. The main construct in PWI PML is the role: an independent thread with its own local data and behaviour. Roles are connected through interactions which are typed, asynchronous buffered message channels. Process*Web* provides a number of standard facilities, in particular there are proxy roles and user roles which are used in communicating with the user interface. The basic design can be illustrated as follows:

1. When a user logs in, a message is sent to their proxy role which returns an HTML page. This page includes a list of all the model instances which the system is running, and a list of the user roles for the model on which the user is currently focusing attention.

2. Through their browser, a user connects to one of the user roles. This involves a message from the browser to the proxy, a message from the proxy to the appropriate user role. The user role has the latest interface, in the form of an HTML page, which it sends to the proxy and thence to the browser.

3. The user now has a model-specific display. Input from a CGI form is sent by the browser, which then waits for a new HTML page in response. The CGI form data is passed through the proxy role and the user role to the actual model role. This will deal with the input, perhaps sending messages to other roles. Then it will send the display representing its new state to its user role, and from there the display is sent to the proxy and then to the web browser.

A proxy role is specific to a user. It copes with the user logging in and out. (In 3 above, the user could logout before receiving the updated display.) It also deals with the user's commands to switch between different user roles.

Each user role is specific to a model role. It stores the latest HTML interface for the model role. A proxy's request for the HTML is often orthogonal to the process execution: a user may simply be browsing the current state of a number of model roles.

The PML interpreter's scheduler is simple. There is a queue of runnable roles, which can include proxy roles, user roles, and model roles. And there is a queue of waiting roles which will become runnable when they receive a message. PWI has facilities which covert an external input, e.g. a message from the browser, into a message in a PML interaction, and so waking a waiting proxy role if required.

3 Feedback

The basic design for proxy roles and user roles is quite simple. However, as Process*Web* has been used and observed over the years extra code has been introduced to deal with specific problems.

3.1 Flicker

On occasions it was found that the HTML page would be updated several times in succession. This can be seen as an advantage, as it confirms that the system is working. However, it can also be seen as an annoying irrelance, when there is no chance of reading the updated output, or as a stupid waste of bandwidth if the user's browser's connection to the Process*Web* server is not very fast.

One cause of this flicker was a build up of messages in the interaction buffer between a user role and proxy, perhaps because the user was viewing a different role. When the proxy did start to handle these messages it would take each one and send it to the browser, causing the flickering update of the display. This was avoided by altering the code so that the proxy informed the user role whether the user was actively interested in its output. If not the user role simply cached the latest display so that it would be available if requested. This enabled the proxy roles to concentrate on handling the data which users wished to view.

The relationship to scheduling is not direct in this example: because of the lack of control over the scheduling, the most economic development approach is to implement, obtain feedback on how the user interface behaves, and then evolve in the light of that feedback. Reviewing the situation it is clear that many evolutionary developments are aimed at alleviating the symptoms rather than tackling core problems. When a user has just supplied input and is waiting for a response the processing done by the proxy and user roles is urgent; when a user is logged out it is a background updating task.

3.2 Wait for Display

As more models were developed, it was discovered that there was a problem in getting interim status messages from computationally intensive tasks. While a display update might be started it did not reach the user's browser and the computationally intensive task hogged most of the processor. Eventually the computationally intensive task finished and then all the status messages got delivered at once.

The solution to this problem involved using a "blocking" interaction. After sending the message to the user role, the computationally intensive task would wait for a resume message on this interaction. This blocks the role, moving it from the running to the waiting queue. The proxy role is also changed. When it sends the status message HTML to the browser, it then sends the required unblocking resume message. There is a cost in this. The proxy role is now more complicated to deal with the extra check, and the model developer has an extra issue to consider.

In this example there is an explicit code introduced to alter the standard scheduling into something which is more acceptable in Process*Web* terms.

4 Discussion

Process*Web* is an E-type system [1]. It becomes part of the world which it models. Its validity depends upon human assessment of its effectiveness rather than its correctness with regard to a specification. In common with many modern systems it incorporates "bought-in" components (e.g. ProcessWise Integrator and a web server) which its developers do not control. Its development shares many of the characteristics of evolution and feedback which stimulated the formulation of the FEAST hypothesis [3]. The system has been continually adapted in response to user feedback to avoid it becoming progressively less satisfactory (Lehman's First Law, Continuing Change [3]). In many cases the sources of dissatisfaction were not observable until the system was used (Uncertainty Principle [2]).

The previous section has taken a particular system function, scheduling, and illustrated that this has had an impact on the evolutionary development of Process*Web*. In brief:

- A scheduling policy is imposed.
- As the system is used, the effects of this scheduling policy are observed and the system is evolved to work around problems.
- The new code which is introduced adds complexity and embeds assumptions about the scheduling policy.

There is definitely a feedback cycle. As the system is evolved, new scheduling effects may be introduced or observed. This illustrates that the evolution of E-type systems, such as Process*Web*, is not exclusively in response to changes in the business process which they support. There are feedback loops where the system architecture gives rise to dissatisfaction, and thus prompts system evolution.

The feedback and evolution described above are typical of a non-compliant system. Often, because the developers have no control over the system function policy, in this case scheduling, its influence is not recognised or ignored. (For example, in response to a problem developers may experiment with a number of coding alternatives until an acceptable result is obtained.) Our vision of a compliant system also has a feedback cycle.

- A scheduling policy is designed, or chosen from a library.
- As the system is used, the effects of this policy are measured. This feedback is used to evolve the scheduling policy.

The most important thing is that the feedback loop is explicitly recognised. Developers are faced with two key questions (adapted from [4]). How to discover what the system is doing? How to structure the architecutre to utilise that knowledge? Our conjecture is that the difficulty of predicting the effects of a software architecture is one of the many reasons why software evolution occurs. A compliant architecture could make a significant change to some of the feedback loops within many E-type systems' development processes. How to measure a compliant architecture and its influence is a key research challenge.[1]

References

[1] Lehman M.M. and Belady L.A., *Software Evolution - Processes of Software Change*, Academic Press, London, 1985, 538 p.

[2] Lehman M.M., Software Engineering, the Software Process and Their Support, in *Software Engineering Journal* vol. 6, no. 5., Sept. 1991, 243 - 258

[3] Lehman M.M., Laws of Software Evolution Revisited, in Montangero C. (ed.) Fifth European Workshop in Software Process Technology (EWSPT'96). Nancy, Oct. 1996, in *Lecture Notes in Computer Science*, vol. 1149, 1996, 108 - 124

[4] Morrison R., Balasubramaniam D., Greenwood R.M., Kirby G.N.C., Mayes K., Munro D.S. and Warboys B.C.. A Compliant Persistent Architecture. to appear in *Software Practice and Experience*, vol. 30, no. 4, 2000.

[5] Process*Web*, http://processweb.cs.man.ac.uk/

[6] Warboys B.C., Kawalek P., Robertson I., and Greenwood R.M., *Business Information Systems: a Process Approach.* McGraw-Hill, 1999.

[1]Research supported by UK EPSRC Grants GR/M88938 and GR/M88945