Laws for Dynamic Systems

Peter Henderson
Electronics and Computer Science
University of Southampton
SO17 1BJ, UK

email: peter@ecs.soton.ac.uk URL: http://www.ecs.soton.ac.uk/~ph

September 1997

Keywords *Reuse Architecture, Reconfigurable Systems, Non-Stop Operation, Active Objects, Reuse Constraints/Laws, Drop-in Components, Distributed Components, Registry, Case Study*

Abstract

A dynamic system is one which changes its configuration as it runs. It is a system into which we can drop new components which then cooperate with the existing ones. Such systems are necessarily built from reusable components, since as soon as the system is reconfigured to use some new components, those new components must reuse the existing, still running, ones.

Design of reusable components in this context is an important problem. We suggest three laws which such reusable components might be required to obey, if dynamic systems are to be effective and to be economically built. We illustrate our conjecture that the laws are effective by describing a generic architecture based on the familiar registry services of OLE or CORBA and by describing a simple point-of-sale system built according to this architecture. We conclude, of course, that some interesting open questions remain. But we suggest that an approach to reuse based on refining the three laws is a promising direction for system architecture to take.

Background

For contemporary business process support systems, the rate of evolution of the components from which they are built has become a dominating factor in the requirement for evolution of the system itself. Either the COTS components go through a release cycle, offering new services which the business needs. Or, new business requirements dictate a need for new or evolved (legacy) components [9]. The need to rebuild the system, or a major part of it, becomes a serious obstacle. Modern architectures, such as OLE and CORBA [15, 16, 17] and client/server architectures in general, address this issue by providing for dynamic reconfiguration of components while the system continues to run. The question arises, what rules should the reusable components in such dynamic systems obey, in order that the system continues to supply an acceptible service throughout periods of change?

In this short paper, we will outline an approach we have taken to the development of dynamic systems from reusable, reconfigurable components. The projects on which this work has been done are developing flexible architectures for business process support systems [8]. Our approach has been to use traditional process modelling languages to model the business requirements and to then map these models onto distributed client/server implementations. For the mapping we have taken a relatively formal approach [6] using the pi-calculus [13, 14, 17]. The pi-calculus provides an effective means of describing the reconfiguration requirements for an evolving system. These formal models have led to the formulation of three laws which we suggest dynamic systems should obey in order to minimise the cost of reusing legacy components, while evolving others. For reasons we will explain, the laws and examples of their application are described informally (or semi-formally) here

.

Dynamic Systems

By a *dynamic system*, we mean a system built from components where the system has the property that new components can be "dropped-into" the system and the system will continue to function, providing continuous and continued service. There are many examples of such systems. Trivially, whenever we run a new program on our existing operating system, we have dropped a new component into it without disrupting it. Similary, in a commercial environment, when we make a new application available to users, we have dynamically evolved the system. In a client/server environment, when we install a new service or release a new client, again we have evolved the available services in a desirable way.

Component Reuse

All these examples are trivial instances of component reuse. They work because of agreed interfaces between the collaborating components (eg files, databases, protocols). Components designed for reuse in these environments follow strict guidelines as to the interfaces which they present. New components adhere to the same guidelines in order to become tolerated members of the community.

In architectures such as OLE, CORBA [15] and Java/ORB/RMI [11] the usual mechanism which allows for the flexibility required by dynamic reconfiguration, is some kind of registry or object request broker. Components offering services publish their offer via the registry. Components requiring services negotiate with the registry and hence, eventually, with the service providers to reach mutually safe collaboration.

It seems that the generic idea of a registry is rather unavoidable for dynamic systems. Hence it become the cornerstone of our suggested architecture. However, the laws we give are characterised in a way which does not presume such an architecture.

Formal Models

We have made use of various formal modelling techniques in the development of the ideas presented here. In particular we have made use of executable specifications [5, 7] and of the pi-calculus [6, 13, 14]. Others have made use of similar formal models for similar purposes [1, 17]. In a seminal paper about the role of formal models in the context of contemporary software engineering, Hoare [10] suggests that the use of formal models to aid understanding, design and abstraction may be a highly cost-effective procedure. We have certainly found it to be the case when considering an architecture for reuse.

However, we have chosen to present our laws informally and to present our examples in a semi-formal way. This is partially to make them accessible to a wider audience. But partially it is because we believe the level of formality chosen here is appropriate to the nature of the ideas presented. However, we have been influenced by the responses we have had over recent months to the presentation of these ideas (to academic and industrial audiences) either as formal models in the pi-calculus or as semi-formal models using the diagrams of the later sections. The formal models tend to elicit discussion on the pi-calculus, whereas the diagrams elicit discussion on reuse, reconfiguration and the generality (or otherwise) of the architecture. This paper is intended to elicit debate of the latter form.

Three Laws

In this section we will give three general laws for dynamic systems and discuss some of their consequences. In the next section we will exemplify an architecture which provides the conceptual framework in which systems obeying the laws can be developed. In the final section we will return to a discussion of the consequences of the laws framed here.

In an earlier section, we have given an informal definition of a dynamic system, phrased in terms of components. Before we can proceed, we need to tighten up the definitions of dynamic system and component.

We want to be able to say that systems are built from components and to draw the meaning of component sufficiently widely that any part of a system can be called a component. So components will have state. They will respond to actions (eg messages, method invocations, events etc) enquiring of or altering their state. But they will also have autonomous behaviour. They will be *active* objects [2, 3, 4]. We have developed the idea that a system can be built from (and usefully described in terms of) a single type of component elsewhere [8].

We generalise the notion of component by saying that a

component supplies services (eg methods, message handlers, event handlers) which may be used by other components.

In these terms, a system is a collection of components which cooperate by each using the services supplied by others. A dynamic system is a system which can be reconfigured (new components added, old oned removed) without having to stop.

We conjecture that a dynamic system will operate safely and effectively if its components all obey the following three laws.

1. A component, added to a system, may not disrupt the behaviour of that system.
2. A component, using the services of another, does so at its own risk and must protect itself from damage.
3. A component offering a service does so at its own risk and must protect itself from misuse.

We will illustrate a system which is built from components which obey these laws as a proof of existence, in the next section. First, let us discuss some of the apparent consequences of the laws.

A consequence of the first law is that a system can choose to completely ignore a new component. That at least would guarantee no disruption.

But such a corollary implies that the system has some behaviour independently of the behaviour of its components. This is not our intention. The system comprises its components and nothing else.

A consequence of the second law is that a component may not rely upon the continued provision of a service supplied by another component. If it did, then the supplier could damage it simply by witholding the service.

A consequence of the third law is that the provider of a service must be fair to its users. Otherwise it is open to a denial-of-service attack, whereby a malicious user is able to deny other users access by locking them out in some way.

There are many such consequences of the laws. Rather than elaborate on more of them here, we will turn to an example which illustrates an architecture which, it is conjectured, supports components which obey these laws.

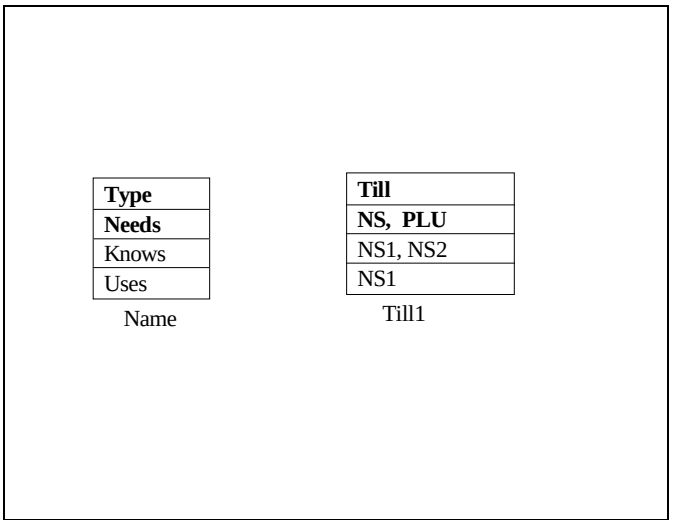

Figure 1 Components: Format (left) and Instance(right)

An Architectural Model

Figure 1 shows the model we are going to use to describe our architecture. On the left of Figure 1 is the format for presenting the details of a component. On the right is an instance of a component which obeys the format.

A component is identified by a unique name, which in the model will be written directly below the component. Thus, in Figure 1, *Till1* is the name of a particular component.

All components have a type, which determines what services they provide. This model does not go down to the level of services. Other components know which services they need by knowing the type of component which supplies them. The type of a component is written in the first (sub-)box of the component model. Thus in Figure 1, the component on the right is of type **Till**.

Each component will need access to a set of service providers. The second box of the component description is therefore a set of component types, called the **needs** of the component. In Figure 1, *Till1* needs to have access to the services provided by a component of type **NS** and a component of type **PLU**.

Each component will, at any point in time, know of a set of other components. This set, called **knows** in Figure 1, will grow as the life of the component extends. Obviously the component's objective in life is to maintain a set of components which it knows, and which supply the

services which it requires. In Figure 1, the component *Till1* knows of the components *NS1* and *NS2*.

Finally, there will be a subset of the components which it actually **uses**. The fourth and final part of the description of a component is this subset. In Figure 1, although *Till1* knows of *NS1* and *NS2* it only uses *NS1*.

To summarise:

A component is known by a **Name**.

A component has a **Type**.

A component has a set called **needs**, which is set of Types being the types of components which it needs access to in order to work.

A component has a set called **knows**, which is set of Names being the names of components which it knows the existence of.

A component has a set called **uses**, which is set of Names being the names of components which it is currently using.

Clearly, there are relationships between the parts of a component. Specifically **uses** will be a subset of **knows** and every component in **knows** will be of a type in **needs**. Moreover, the relative values of these parts can be used to describe the evolving state of a component. For example, it may be in a state of using components which satisfy its needs, or it may be in the state of still needing components. We will not go into these aspect further here, since it takes us away from our objective of discussing the three laws.

An Example

We are going to be very specific and show an example taken from a retail, point-of-sale support system. The system comprises tills (or cash-registers in the UK) at which customers check out goods. The tills need access to a component which provides price-lookup (PLU), among many other services. The system is organised around a registry (NameServer, NS) at which the providers of services register their availability and from which the users of such services learn the identity of providers.

The semi-formal notation introduced in the last section and exemplified in Figure 1 has been designed for visual presentation. Specifically, the presentation is an animation using Powerpoint Slideshow. The dynamic nature of the system is apparent from the features of this animation. New components appear on the screen among an unchanging backdrop of already configured components. As the story unfolds of how each component configures and reconfigures itself, the individual attributes of each component change one at a time. The visual cues which this gives are ideally suited to the human eye and everyone sees instantly where their attention should be.

The formal properties of the architecture are important, but presenting them formally to scientists and engineers is less effective than the animation at eliciting the necessary debate about the design. The architecture is simplfied because of the existence of the formal model. The animation serves as an ideal overview of the formal model. When eventually we get to very hard questions, or to the need for very precise statements, then it is time to turn to the mathematics for help.

We can not present the animation here (unless you are reading this on the Web). The next best thing is to choose some of the major stages and to go through the sequence of events between stages. This is what we will now do for out point-of-sale system.

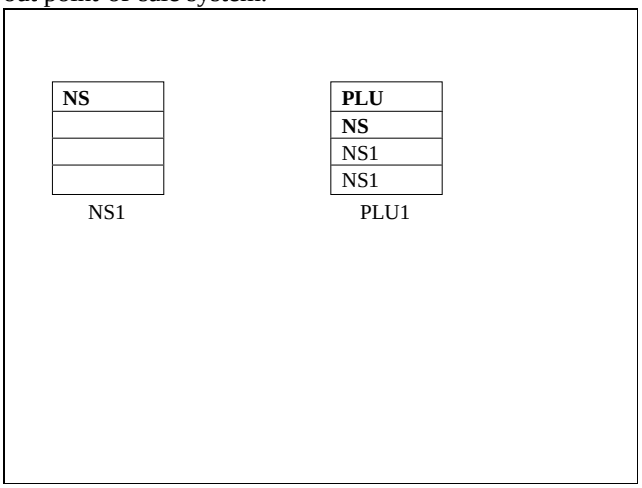

Figure 2 A NameServer (NS1) component and a Price Lookup (PLU1) component

The point-of-sale system is built around a registry component (of type **NS**, NameServer) which provides an introduction service. Figure 2 shows the NameServer *NS1* as it would have appeared at the beginning of time. In Figure 2 we assume a Price Lookup component *PLU1* (of

type **PLU**) has just arrived. It already knows of *NS1*. This prior knowledge of something (a registry, usually) is assumed of all components. It is not the only architectural choice we could have made.

The component *PLU1* registers with *NS1*. Figure 3 shows the system state which ensues.

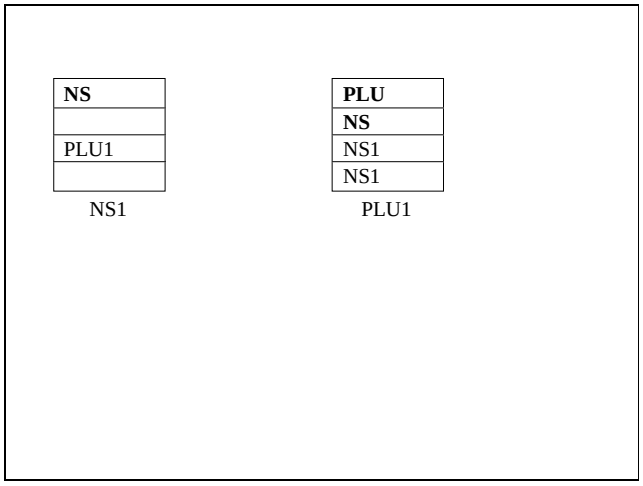

Figure 3 The Price Lookup component registers with the NameServer

Now the system is ready for tills to come on line. Figure 4 shows the state which we have as the first till component comes into existence.

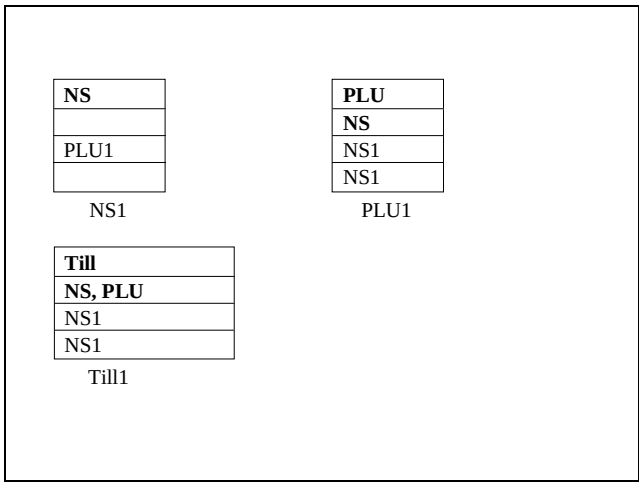

Figure 4 A Till arrives, it requires a PLU

The obvious sequence of events takes place. The Till, *Till1* knows only of the NameServer *NS1*. It requests a component of type **PLU**. The NameServer will supply one or more names. It only knows of *PLU1*, so we assume it supplies that. Now we move to the state shown in Figure 5.

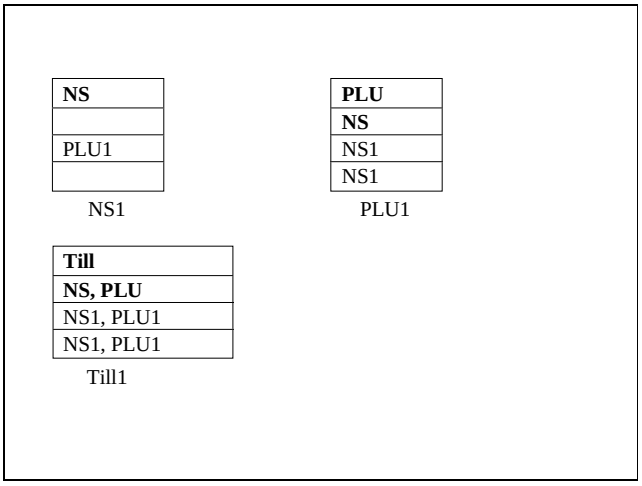

Figure 5 The Till learns of PLU1 from NS1 and connects to it

You can imagine how the animation proceeds. A second Till arrives and connects to *PLU1*. A second PLU arrives (call it *PLU2*) and registers with *NS1*. Now *PLU1* fails, but the system recovers because both tills reconfigure to use *PLU2*.

We go on. What if the NameServer fails? Well, to start with its OK. Until a PLU fails. There are many scenarios possible. Maybe the Tills must wait until a new NameServer arrives and the PLUs register. Maybe the Tills took the precaution of finding out about the second PLU service and can reconfigure without a NameServer. Maybe the Tills know about each other and one of them knows of an alternate PLU service. Each of these scenarios demonstrates a different aspect of three laws.

Figure 6 shows a typical intermediate state where the various components have done something to protect themselves against failures.

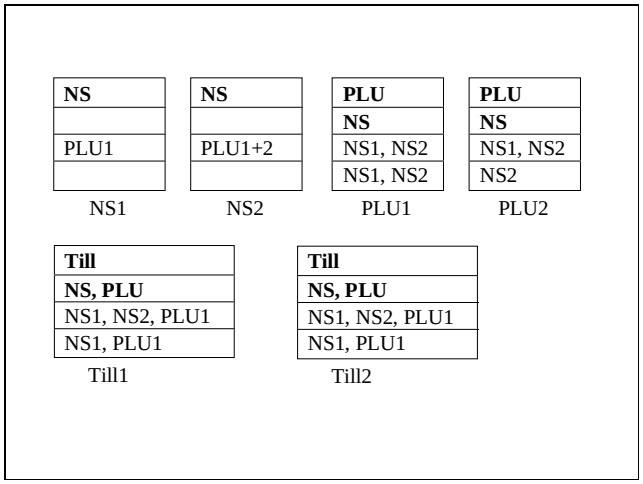

Figure 6 A typical intermediate state where components have done something to protect themselves against failures

But the interesting scenarios are not these fault tolerant ones, even though they show reuse of legacy components. The interesting scenarios are those where we go through a component evolution (typical of COTS) where eventually we replace an old component by a new one, other components still protecting themselves against damage. We shall discuss such scenarios in the context of our three laws in the next section.

Discussion

Our conjecture is that, reusable components must obey constraints if they are to be good members of a community comprising a dynamic system. Particulary, we wanted to address reuse in the context of dynamic systems which we defined as collections of components which cooperate by supplying services to each other. We enunciated three laws which we conjectured reusable components could choose to obey. We suggested that a system built from components obeying these laws would have the desirable properties of effective dynamic reconfiguration, that is change of functionality without loss of service.

To what extent have we been able to support this conjecture? The architecture described in the earlier sections is a proof of existence of a set of components which obey the laws and apparently deliver the stated properties.

The first law is illustrated by arrival of components of type **Till** and **PLU** which, on arrival, simply register with or make a request of the NameServer. This behaviour can not disrupt the behaviour of the system per se, although it is dependent on the ability of the other components to protect themselves against instances of such components which are, in some sense, undesirable members of the community. The power of the first law comes from the fact that it dictates that old components must take an active part in the decision to collaborate with new ones.

The second law is illustrated by the Till surviving the loss of a PLU. Similarly, it is illustrated by the Till surviving the loss of both a PLU and a NameServer. The Till has made use of these services at its own risk and has not been damaged by their loss. It would also be illustrated by the more interesting scenario (too elaborate for inclusion here) of a Till surviving the replacement of an old PLU service by a new one which it subsequently finds deficient. Obeying the second law it would retain the ability to reconfigure back to using the old service.

The third law is illustrated by the PLU protecting itself against a denial-of-service attack, where a malicious Till has tried to lock other users out. A PLU obeying the third law would not allow this. A corollary is that the PLU service must be multithreaded or connectionless or in some way able to deliver the fairness required by the third law.

However, this proof of existence, such that it is, only goes some way towards supporting our conjecture.

It is possible that the laws are not even internally consistent. Our examples only show that the suggested architecture appears to be consistent and appears to obey the laws.

Whilst it may be that any system which has the properties we require will obey the laws. it is unlikely that the laws are complete in the sense that any system which obeys the laws has the properties which we require.

The laws are probably not independent. Moreover, they may even be redundant in that one may be a consequence of the others.

Notwithstanding these difficult questions, the three laws do seem to have some utility in scoping the architectural

debate which it is necessary to have in order to design dynamic systems which are built from reusable legacy or COTS components. This we have exemplified by capturing a description of a common generic architecture , the registry centred architecture common to OLE, CORBA and client/server systems. We have taken the debate in only one of many possible directions in this paper. For example, we have made the assumption that the registry only knows of components which register themselves. An alternative architecture might remove the need for any action on the part of a new component. Would such an architecture be more robust?

It will be interesting to discover, as these laws become more refined and as the systems we build test their utility for scoping the design of reusable components and continuously evolving systems, the extent to which this initial formulation captured (or failed to capture) the essence of what is, after all, just the way we build contemporary systems.

Acknowledgments

I am indebted to many colleagues in ICL who have encouraged me to develop these ideas over recent years and have sponsored the projects in which they were developed. In particular Simon Hayward, Ed Parton, Alun Roberts, Graham Pratten, Bob Snowdon and Peter Wharton have contributed, at different times, to the ideas presented here. I am indebted to my colleagues in Southampton who provide a stimulating environment in which to argue one's case. And I am indebted to my colleagues in IFIP WG 2.3 who are never satisfied with anything, and so present me with an eternally reusable challenge.

This work was carried out as part of EPSRC grants GR/J08928, GR/K08116, GR/K83014.

References

[1] Allen R.J, Remi Douence and David Garlan Specifying Dynamism in Software Architectures *Workshop of Foundations of Component Based Systems*, Zurich, 1997, see http://www.cs.iastate.edu/~leavens/FoCBS/index.html

[2] Birrell, A.D Greg Nelson, Susan Owicki, and Edward P. Wobber. Network Objects. Software Practice and Experience, 25(S4):87-130, December 1995. Also appeared as SRC Research Report 115, see http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-115.html

[3] Brown, Marc H and Marc A Najork (1996) Distributed Active Objects *Computer Networks and ISDN Systems*, 28:1037--1052, May 1996. (*Proceedings of the Fifth International World Wide Web Conference*, Paris, France, May 6-10, 1996). Digital Systems Research Center, Report 141a, http://www.research.digital.com/SRC/bibindex/tmp/30547.html

[4] Cardelli, Luca (1995) A Language with Distributed Scope. *Computing Systems*, Vol 8, No1,Jan 1995.

[5] Gravell, A. and P. Henderson, Executing formal specifications need not be harmful, *Software Engineering Journal*, vol. 11, num 2., IEE (1996).

[6] Henderson P Formal Models of Process Components, *Workshop of Foundations of Component Based Systems*, Zurich, 1997, see http://www.cs.iastate.edu/~leavens/FoCBS/index.html

[7] Henderson, P Functional Programming, Formal Specification and Rapid Prototyping, *IEEE Transactions on Software Engineering*, Vol.12, No.2, pp.241-250, 1986

[8] Henderson, P & Pratten, G.D. POSD - A Notation for Presenting Complex Systems of Processes, in *Proceedings of the First IEEE International Conference on Engineering of Complex Systems*, IEEE Computer Society Press, 1995

[9] Henderson, P Software Processes are Business Processes Too *3rd International Conference on the Software Process*, IEEE Computer Society Press, Oct 1994

[10] Hoare C.A.R How did Software get to be so reliable without proof *Keynote address at the 18th International Conference on Software Engineering*. IEEE Computer Society Press, 1996. see also *http://www.comlab.ox.ac.uk/oucl/users/tony.hoare/publications.html*

[11] JavaSoft Java RMI specification 1996, see http://www.javasoft.com

[12] Magee J and Kramer J Dynamic Structure in Software Architecture *Proceeedings of the ACM Conference on Foundations of Software Engineering*, 1996

[13] Milner, Robin The Polyadic pi-calculus: a tutorial *International Summerschool on Logic and Algebra of Specification*, Marktoberdorf, 1991. see also http://www.dcs.ed.ac.uk/lfcsreps/EXPORT/91/ECS-LFCS-91-180/index.html

[14] Milner, Robin Elements of Interaction: Turing Award Lecture *Communications of the ACM*, Vol 36, No 1, January 1993

[15] Object Management Group Common Object Request Broker: Architecture Specification see

[16] Sullivan K and Knight J.C Experience Assessing an ArchitecturalApproach to Large Scale Reuse *Proceedings of ICSE-18*, 1996 IEEE Computer Society Press

[17] Sullivan K, Socha J and Marchukov M Using Formal Methods to Reason about Architectural Standards *19th International Conference on Software Engineering*, Boston, IEEE Computer Press, 1997