Modelling Architectures for Dynamic Systems

Peter Henderson

ABSTRACT A dynamic system is one that changes its configuration as it runs. It is a system into which we can drop new components that then cooperate with the existing ones. We are concerned with formally defining architectures for such systems and with realistically validating designs for applications that run on those architectures. We describe a generic architecture based on the familiar registry services of CORBA, DCOM and Jini. We illustrate this architecture by formally describing a simple point-of-sale system built according to this architecture. We then look at the sorts of global properties that a designer of applications would wish a robust system to have and discuss variations on the architecture which make validation of applications more practical.

1 Introduction

The advent of ubiquitous computing, where everything is connected to everything else, has created a new challenge for Software Engineering and for Software Reuse in particular. It is now increasingly important that software components are designed for a life of constant change and frequent reuse.

With everything connected to everything else, systems are necessarily subject to dynamic change. You can't stop the whole world just to plug in a new component. Components need to be as nearly plug-and-play as possible. Flexible architectures such as Jini [35], are making the evolution of dynamic systems possible. The question is, how do you design for such architectures and how do you design components which will survive a lifetime of use and reuse even though their environment and the expectations which their users have of them, are constantly changing?

1.1 Dynamic Systems

In [16] dynamic systems are described as being built from components and having the property that, a new component could be added to a running system at any time and the system would embrace its contribution without having to stop. It is the requirement that the system can evolve by accretion, without ever having to stop, that leads us to call the sys-

tem"dynamic". The consequences for component reuse are dramatic. Components will be reused in ways that were not imagined by their original designers. In [16] we addressed the issue of who would be to blame if the consequence of adding a new component was that something broke.

In this paper, we formally describe some of the issues which arise for the developer of dynamic systems, not least of all the evolution of functionality in an incremental way. We do this by introducing an elementary architecture modelling language, ARC [17], which allows for experimentation with alternative architectural designs and for the validation of these designs using state-space search. In particular, ARC models can be compiled to run on the SPIN model-checker [21]. The ARC modelling paradigm, it is conjectured, is simple enough to allow many experiments to be performed quickly with modest cost and yet powerful enough to describe a range of practical architectures and generate valuable insight into their properties.

1.2 The Context of Constant Change

We are concerned with dynamic systems in the context of constant change, where the system supports a business process which is constantly needing to be changed to match the rapidly moving marketplace. We wish to explore architectures which will allow the incremental enhancement of the system without having to be stopped for upgrade. Consequently we are concerned with issues of reconfigurability, where new components can be added to the system which then embraces the new services which they offer. We are less concerned with the removal of old components in that we anticipate architectures which will allow such components to gradually become obsolete and eventually retire.

However we are concerned with issues of survival. We will articulate scenarios in which the system can survive despite the fact that some components fail. One way of looking at this issue is to characterise the interaction between a system and its environment as a two-person game [1]. The moves made by the system are to maximise the number of components which can operate. The moves of the environment are to damage key components with the intention of preventing as many components as possible from operating successfully. We show how our modelling paradigm lends itself to this metaphor.

2 Models of Dynamic Systems

In order to be able to make precise statements about alternative architectural proposals we need to use a language which has a precise meaning and which operates at a level of abstraction appropriate to the kinds of reasoning which we wish to perform. There is a choice of paradigms. Many

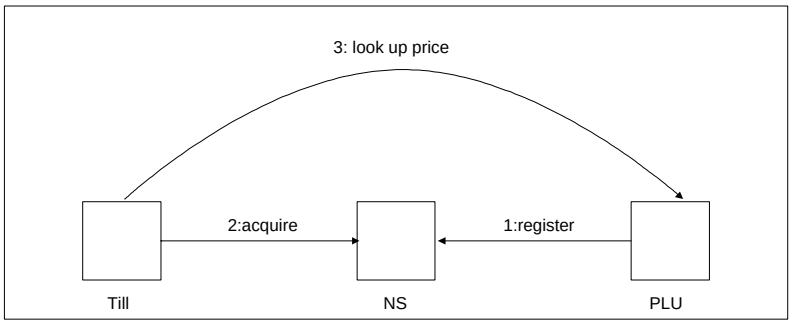

FIGURE 1. A UML Collaboration Diagram

architecture modelling languages base themselves on a message-passing, process-oriented view. Examples are Wright [2, 3, 32], Darwin [27, 28] and more recently FSP/LTSA [29]. Others, such as Rapide [24, 25, 26] take an event based approach where events are specified by condition-action rules. This is the approach we will take in ARC. Architecture languages concern themselves with structure and behaviour [36]. We are, of course, concerned with both here. But, in a dynamic system, structure is dynamic and so structure merges into behaviour.

2.1 The ARC Notation

Our conjecture is that our modelling language is appropriate to the design of reusable components for dynamic systems, because it operates at a level-of-abstraction that allows reasonably large systems to be modelled, but still allows a useful degree of validation of the models in a cost-effective manner.

The modelling paradigm is influenced by the collaboration diagrams of UML [9]. These diagrams are a variety of Object Interaction Diagram, where the behaviour of a (scenario) from a system is depicted. In collaboration diagrams (see Figure 1), objects (rather than classes) are shown along with the messages which pass between them. The objects are usually boxes and the messages are arrows. The sequencing is shown by numbering the messages. The reader can then follow a scenario by reading the messages in order. Designers use such a diagram to first convince themselves, then others, that they have a valid behaviour.

Figure 1 shows an example of a UML collaboration diagram and also serves to introduce the example which we will use throughout this paper both to introduce ARC and to consider alternative architectures. Figure 1 shows an EPOS (Electronic Point-of-Sale) system. It shows three objects: *Till* is the (hardware and software) component where customers' purchases are scanned and paid for; *NS* is the Name Server which (in this client-server architecture) acts as the registry for objects offering and requiring services; and *PLU* is the Price Look Up component which offers the service

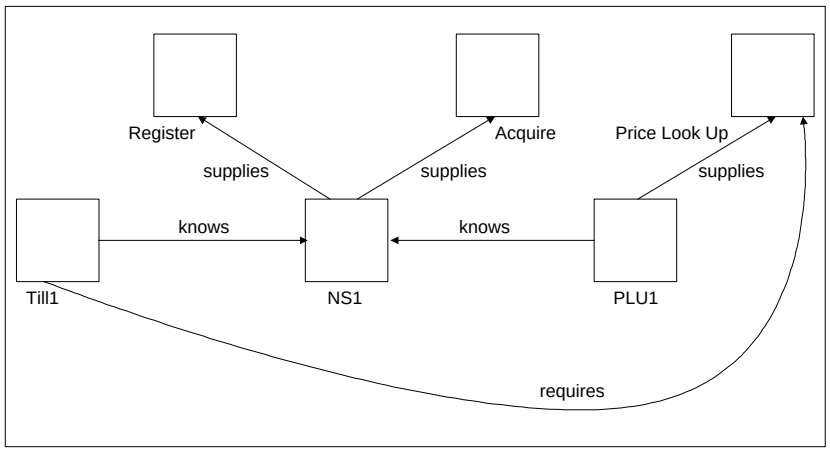

FIGURE 2. An ARC Diagram

of supplying prices for purchased items.

The scenario depicted in the UML diagram of Figure 1 shows a sequence of three operations: first the *PLU* invokes a *register* operation on *NS* to register its availability as a supplier of the *PriceLookUp* service; then *Till* acquires from *NS* the name (*PLU*) of the supplier of this service; finally the *Till* invokes a look-up-price operation on the *PLU*.

Although inspired by the collaboration diagrams of UML, our paradigm uses a slightly higher level of abstraction. Rather than show messages, we show relationships or associations, between objects. The implication is that, if an appropriate relationship exists between two objects, one may have access to the services of the other. We will illustrate this in detail in what follows. The behavioural aspect of the system that we will then be able to illustrate is the configuration and reconfiguration of those relationships as, in a dynamic system, components first join and then acquire relationships with other components which they intend to use.

In ARC we use the terms component and object interchangeably. We think of objects or components as having state, behaviour and autonomy. That is, they are active, as if they were servers or clients. Figure 2 shows the state of a system in ARC diagrammatic form. There are six components (*Till*1, *PLU*1, *NS*1, *Register*, *Acquire* and *PriceLookUp*) and three relationships (*knows*, *supplies* and *requires*). The diagram depicts the state in which, among other things, *Till*1 requires *PriceLookUp*, *PLU*1 supplies *PriceLookUp* and while *Till*1 does not yet know of *PLU*1, it does know *NS*1 which in turn knows *PLU*1. *NS*1 is, of course, the Name Server in this distributed system. *Till*1 will ask *NS*1 for the name of a component which supplies *PriceLookUp*, and as a consequence the configuration will change dynamically to add the relationship that *Till*1 knows *PLU*1.

In practice, the ARC diagrams become too cluttered to express realistic

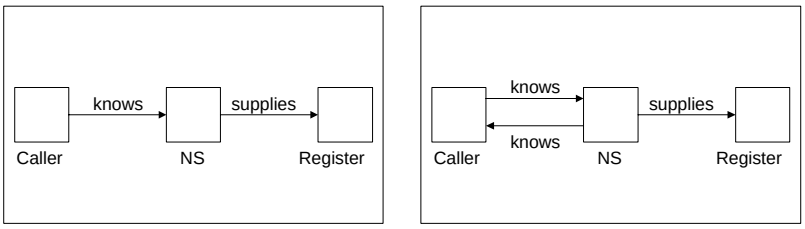

FIGURE 3. The Register Action

scenarios, so we use them only to illustrate partial states. They come into their own when a group of engineers are designing a new solution on a whiteboard, because changes to the solution are quickly understood by all the participants. But for formal presentation we use a textual form to capture the full meaning of any situation. That is what we shall use here.

The ARC textual notation is based on logic, and in particular on the use of logic in Prolog strongly influenced by conceptual modelling [4]. A similar use of notation has recently been adopted in Alloy [22].

The state depicted in Figure 1 would be expressed by the conjunction

$$knows(Till1, NS1)\&knows(NS1, PLU1)\&supplies(NS1, Register)\&$$
$$supplies(NS1, Acquire)\&supplies(PLU1, PriceLookUp)\&$$
$$requires(Till1, PriceLookUp)$$

This is how we describe a state, as a conjunction of (usually) binary relations. Next we describe Actions which will enable us to move from state to state. We use Condition-Action rules. Figure 3 shows a diagrammatic form of a rule, with the condition to be met depicted in the left-hand box and the state to be moved to depicted in the right hand box. What the Action in Figure 3 depicts is the act of registering with a Name Server NS. The $Caller$ knows NS initially, and in the eventual state NS knows the $Caller$.

Putting this Action into textual form, we have

$$register(Caller, NS) =$$
$$knows(Caller, NS)\&supplies(NS, Register)$$
$$\rightarrow +knows(NS, Caller)$$

Thus we define actions, which we give names to, which have a side-effect of adding and deleting relationships. Actions have parameters. The addition and deletion of relationships is denoted by $+$ and $-$ signs just in front of the relationship name. An example of relationship-deletion would be the reverse of the Register operation shown in Figure 4.

$$deregister(Caller, NS) =$$
$$knows(Caller, NS)\&knows(NS, Caller)\&supplies(NS, Deregister)$$
$$\rightarrow -knows(NS, Caller)$$

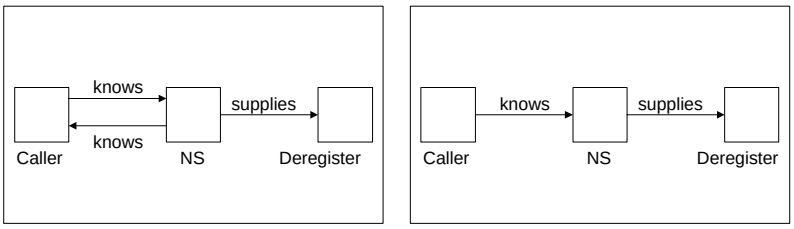

FIGURE 4. The deRegister Action

A more complex Action is shown in Figure 5. This is the Acquire action that also involves a Name Server. It is the way that components obtain knowledge of others that supply services which they require. The Action depicted in Figure 3 has the meaning

$$
\begin{aligned}
&acquire(Caller, NS, Service) = \textbf{exists } Object.\\
&\quad knows(Caller, NS) \& knows(NS, Object) \& supplies(NS, Acquire) \&\\
&\quad supplies(Object, Service) \& requires(Caller, Service)\\
&\quad \rightarrow +knows(Caller, Object)
\end{aligned}
$$

You can see how this would match a state in which, for example

$$knows(Till1, NS1) \& knows(NS1, PLU1)$$

and $NS1$, $Till1$ and $PLU1$ are as previously described. So if this action is performed on that state, we would move to a state in which $Till1$ knows $PLU1$, an obviously desirable state of affairs.

This is mostly all there is to ARC. In the formula for *acquire*, the component $Object$ has a particular status. It is not a parameter of the operation. It is a local variable, which can match any component that satisfies the relational structure in which it is involved. In logical terms, it is existentially quantified with scope the condition and action parts of the rule. In addition to the logical structures which we have exhibited here, we allow explicit negation, disjunction, implication and universal quantification. Negation could have been used in the formula for *acquire*, for example, to strengthen the condition in such a way as to ensure that the $Caller$ did not acquire something which supplied a service which was already supplied by some component which it already knew (add $\neg(knows(Caller, Object1) \& supplies(Object1, Service))$ to the condition).

2.2 Validation of Models

The models we have made are particular forms of finite state machines, with the states represented by a particular edge-coloured graph, where the nodes are Components, the edges are Relationships and the colours are the

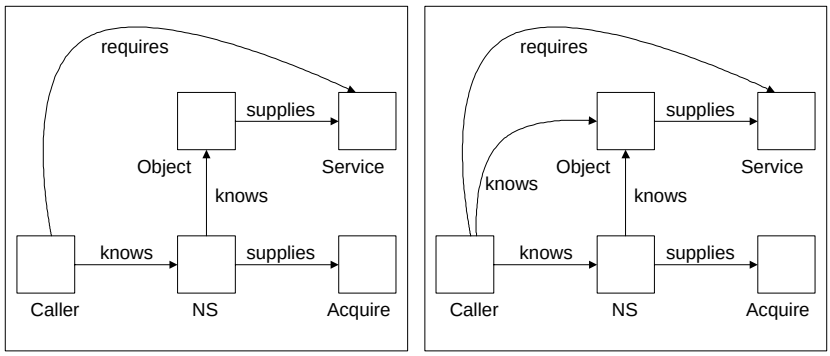

FIGURE 5. The Acquire Action

actual Relations. Transitions between states are accomplished by Actions, which add and remove edges from the graph.

Consequently, validation of the models can be accomplished by various finite state machine checking capabilities. In particular, model checking can be used [6, 13, 21, 22, 29]. It is also straightforward to build animations of the models, and this is an effective way for a group of engineers to persuade themselves that their design is complete and consistent, and to look at the consequences, for example, of component failure.

As an example of validating a model, consider the example we have used throughout this introduction to ARC. In the simplest scenario, we might begin in the state

$$knows(Till1, NS1)\&knows(PLU1, NS1)\&supplies(NS1, Acquire)\& \\ supplies(NS1, Register)\&requires(Till1, PriceLookUp)\& \\ supplies(PLU1, PriceLookUp)$$

Now the reader will realise that the sequence of Actions

$$register(PLU1, NS1); acquire(Till1, NS1, PriceLookUp)$$

will move us to a situation where, in addition to the above state, we also have the following relationships

$$knows(NS1, PLU1)\&knows(Till1, PLU1)$$

Figure 6 shows the ARC validation tool which supports various types of state space search. The model developed in this section has been presented to the tool which displays three panels each containing a list. The user chooses to instantiate a small number of objects of each type, in this case one Name Server ($NS1$), two Tills ($Till1$ and $Till2$) and two PLUs ($PLU1$ and $PLU2$). The list of Actions displays (in alphabetical order) just those which are effective in that their condition part evaluates to true and their action part will actually change the state. The State list comprises terms

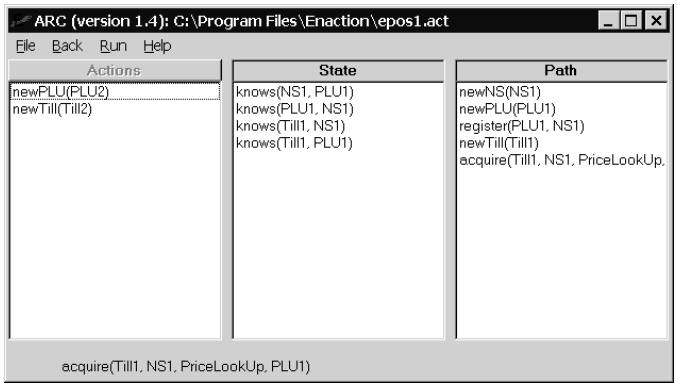

FIGURE 6. The ARC evaluation tool

in the conjunction which describes part of the (graph representing the) current state in which we have expressed an interest. Selecting an action from the Action list applies it to the current state and hence progresses to the next state. The actions which have been invoked so far are shown (this time in sequential order) in the Path list. Various methods of searching the state space are provided [17]. The most elementary is just to move forward and backwards taking different branches manually. The Back button on the menu bar moves the user progressively back through the path they have followed.

ARC models can also be translated into Promela [21] in a very straightforward way and executed on the SPIN model-checker. Every relationship of the form $rel(obj1, obj2)$ (that is, every edge potentially in the graph representing the state) is represented as a Promela (bit) variable. Adding the relationship to the state corresponds to setting this variable to true, removing the relationship to setting it to false. Experiments have shown that ARC and SPIN generate the same state machines. Translating to SPIN has the advantage that we can make use of SPIN's mature model-checking capabilities, particularly its performance and its ability to check temporal properties expessed in LTL.

3 Architectures for Reuse

The client-server architecture which we have used to illustrate our modelling language is an example of a flexible architecture designed for reuse of (Services supplied by) Servers. We have shown that it is able to support the elementary kind of reconfiguration required by the initial marriage of clients to servers. And we can show that it is tolerant of some types of failure and incremental change.

3.1 Survival

Consider the following kind of attack on the client-server architecture

$$\begin{aligned} &break(Object) = \\ &\quad \textbf{all}\ Service.supplies(Object, Service) \rightarrow \\ &\quad -supplies(Object, Service) \end{aligned}$$

Clearly, if we execute $break(PLU1)$ then this can be fixed by the system performing $acquire(Till1, NS1, PriceLookUp)$ again, which will locate $PLU2$, assuming it has registered with $NS1$. Breaking $NS1$ with $break(NS1)$ is a little more serious, but not immediately. The system continues to function. It runs into trouble after $break(PLU1)$, for now $Till1$ cannot find $PLU2$. Unless of course $Till1$ had had the foresight to prepare for this eventuality by acquiring $PLU2$ even though, having $PLU1$, it didn't strictly need it. But of course, eventually the loss of $NS1$ is more serious.

The semantics of creation of new objects gives a telling insight

$$\begin{aligned} &newPLU(PLU) = \\ &\quad true \rightarrow +knows(PLU, NS1); +supplies(PLU, PriceLookUp) \end{aligned}$$

The other object creating definitions are similar. In this system, every new object comes into existence knowing the name of the same single registry $NS1$. When $NS1$ dies, the system can only deteriorate.

But even here, there is a solution. It has to do with where the initiative for performing actions is assigned. In the model, we have purposely not assigned the actions to the objects. But we should, because we want objects to be autonomous and active. The reason we haven't done this in the model is that we don't want to decide early either who has the initiative or what their goal is. But suppose that all objects know how to invoke *acquire* on objects which supply that service and that their objective (goal) is to acquire as many instances of the objects which supply services which they may be able to use. Then, if $NS2$ is created and registers with $NS1$, all the objects which know $NS1$ can now acquire $NS2$ and thus increase their chances of survival.

Note that formally our architecture requires one of two changes. Either we weaken the condition on *acquire* to omit $requires(Caller, Service)$ so that objects can acquire anything, whether they need it or not. Or, we strengthen the requirements of all objects to include $+requires(Object, Acquire)$.

3.2 Incremental Change

This leads to another consideration of how systems evolve, rather than just survive. Suppose that we plan to upgrade our EPOS system with a new service. For the sake of argument let us assume it is a Loyalty scheme whereby

the system identifies the customer at the point-of-sale and offers bespoke services (such as targeted coupons). We will run through one scenario which illustrates this happening.

First we have a new Loyalty Server,

$$newLS(LS) =$$
$$true \rightarrow +knows(LS, NS1); +supplies(LS, Loyalty)$$

Then we have a new Till

$$newLSTill(LSTill) =$$
$$true \rightarrow +knows(LSTill, NS1);$$
$$+requires(LSTill, PriceLookUp); +requires(LSTill, Loyalty)$$

The interesting question is, if we create $newLSTill(Till1)$ say, does it inherit the existing configuration of $Till1$. The formal model says it does. If that is not what we intended, then we need a tighter description.

Suppose we define

$$exists(Object) =$$
$$\textbf{exists}\ relationship.(relationship(Object, Something)\textbf{or}$$
$$relationship(Something, Object))$$

then we can strengthen the precondition on $newLSTill$ to be $\neg exists(LSTill)$. If however, what we require is to genuinely model the fact that the old $Till1$ and the new $Till1$ actually share something other than a name, for example that they share the same hardware, then we need to separate the objects which represent the Till application from those that represent the Till hardware. This can be done, but we will not go into it here.

Of course, in all practical cases we must realise that systems are implemented at different levels and we will need to model components at different levels of abstraction. Figure 7 shows how this is done. In a high level model, the relationship $knows$ will be stored explicitly. In a refinement of that model, the relationship $knows$ will be derived from other relationships (stored or derived). Figure 7 shows how the $PLU1$ and the $NS1$, in separate environments (processes, name spaces, machines etc) come to know each other by a conjunction of relationships, set up presumably by more primitive actions than $acquire$.

$$knows(A, B) =$$
$$localKnows(A, TX1)\&localKnows(B, TX2)\&$$
$$globalKnows(TX1, TX2)$$

3.3 Loosely Coupled Components

The architectures we are trying to describe to support reuse in the context of constant change, with its consequent need for dynamic reconfiguration is

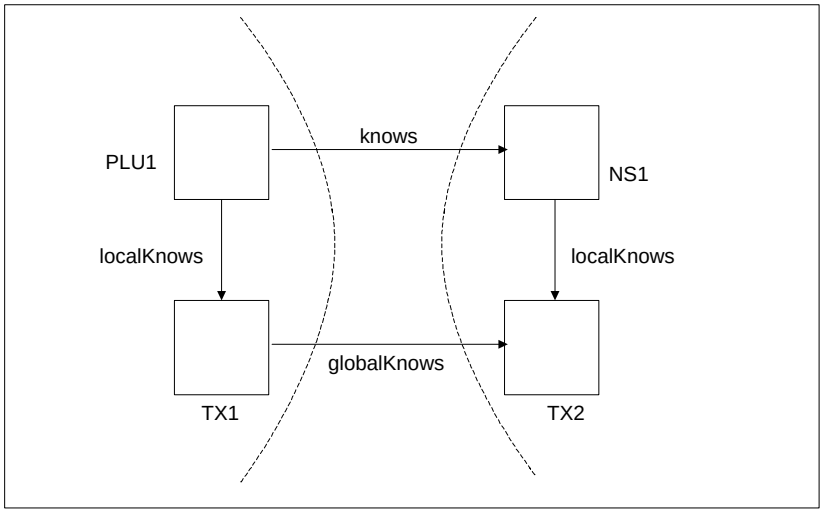

FIGURE 7. Separation of Levels

leading us towards increasingly loosely-coupled components. The architecture which we have used as our example, the client-server architecture, has some of this looseness of coupling. Dynamic binding is achieved through the use of registry services such as the Name Server which we have used. As an example of something more loosely coupled consider the following architecture which is a development of the client-server. We don't have specialised Name Servers. Rather, every object is a Name Server. This is achieved by ensuring that every object supplies both *Register* and *Acquire*. Now, on creation, every object must know the name of some other object, but that doesn't have to be always the same object. Given the initiative to seek out as many new objects as it can, a new object can increase substantially its chances of being able to survive and continue to function, notwithstanding attacks from elsewhere.

3.4 Its all a game

We can characterise the fight for survival of a system, or perhaps more accurately the components within the system as a 2-person game. Imagine that the two players are the System itself and the Environment. The System can make a move comprising a sequence of actions, thus moving to a desired state, whereupon it yields. The Environment can then make a move which we presume will break something. The Environment wins if the System gets into a position from which it can not recover to a position which it is required to achieve.

Restricting the Environment to a single break action is a modelling choice, but it does allow us now to specify an interesting property of a Sys-

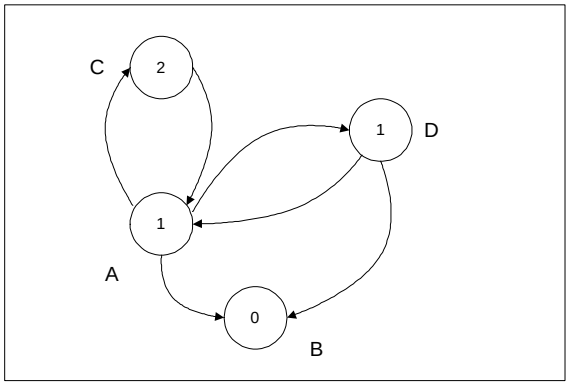

FIGURE 8. Positions in a game of Survival

tem state. The property is an integer which counts the number of moves the System is away from losing. Consider Figure 8. This shows a common situation in the game of survival. Each node in the diagram represents a state of the system and the integer in the circle is the number of moves the System is away from disaster. The game starts at node A. If the System is astute enough always to move to C, whenever it is at A, then the System survives. If it ever moves to D, then there is the chance that the Environment will win by moving to B. You can see how this metaphor reasonably captures the notation of survival for a dynamic system. We hope to show that it also reasonably captures the notion of incremental change and improvement as the System moves further away from zero. We expect that this will require a considerably more complex numbering scheme. We are investigating whether or not we can develop state models based on the game representation schemes devised by Conway [7].

4 Conclusions

We have concerned ourselves with the formalisation of dynamic systems, which we have characterised as systems of components that need to reconfigure themselves in order to respond to changes in the requirements upon them. We have shown how systems based on registry services are basically appropriate to this problem and have suggested some refinements to this architecture, specifically generalising the reasons why any component might store information about another. Another extension is to make every component (at a certain level) able to provide the capabilities of a registry. We have introduced an architecture modelling language, ARC, which adopts the paradigm of modelling systems as objects and relations. This leads to an elementary behavioural description language which we have shown to be powerful enough to describe the systems which we wished to discuss.

Validating the properties which we have proposed for solutions is possible using state-space search. We have a tool for doing this which we described briefly. More details, and tool itself, are available on the web [17]

Further work on architecture modelling is ongoing. In particular we are building models of MQM [8], of Jini [35] and of the Ambients [5] paradigm as well as showing whether or not ARC can model most of the things that other architectural modelling languages can. Of course, theoretically it is possible to show that ARC can represent anything but we are more concerned with the practical use of the paradigm by software architects and software engineers in practice in real industrial scale tasks. We are confident that this objective will be achieved.

5 References

[1] Abramsky, S and G McCusker, "Game Semantics", *see http://dcs.ed.ac.uk/abramsky*

[2] Allen R.J and D Garlan, "A Formal Basis For Architectural Connection", *ACM transactions on Software Engineering and Methodology*, (July 97)

[3] 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

[4] Boman, M, J.A. Bubenko, P. Johannesson and B Wangler, "Conceptual Modelling", *Prentice Hall*, (1997)

[5] Cardelli, Luca "Abstractions for Mobile Computation" *Microsoft Research Technical Report MSR-TR-98-34* available at research.microsoft.com (1998)

[6] Clarke E.M et al, "Model Checking and Abstraction", *ACM Transactions on Programming Languages and Systems*, Sept. 94

[7] Conway, J, "On Numbers and Games", *Academic Press*, 1976

[8] Dickman, A, "Designing Applications with MSMQ", *Addison Wesley* 1998

[9] Fowler, M, "UML Distilled - Applying the standard Object Modelling language" *Addison Wesley*, 1997

[10] Garlan, David et al "Architectural Mismatch, or, why it's hard to build systems out of existing parts". *ICSE* 1995

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

[12] Gray, David N et al, "Modern Languages and Microsoft's Component Object Model", *Communications of the ACM*, Vol 41, No 5, 1998

[13] Grumberg, O and D Long, "Model Checking and Modular Verification", *ACM Transactions on Programming Languages and Systems*, May 94

[14] Heimdahl, M and Leveson, N "Completeness and Consistency in hierarchical state-based requirements" *IEEE Transactions on Software Enginerring*, 1996

[15] 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

[16] Henderson, Peter. "Laws for Dynamic Systems", *International Conference on Software Re-Use (ICSR 98)*, Victoria, Canada, June 1998, IEEE Computer Society Press

[17] Henderson, Peter. "ARC: A language and a tool for system level architecture modelling", July 1999, see http://www.ecs.soton.ac.uk/ ph/arc.htm

[18] Henderson, Peter and Bob Walters, "System Design Validation using Formal Models" *Proceedings 10th IEEE Conference on Rapid System Prototyping, RSP'99*, IEEE Computer Society Press, 1999

[19] Henderson, Peter and Bob Walters, "Component Based systems as an aid to Design Validation" *Proceedings 14th IEEE Conference on Automated Software Engineering, ASE'99*, IEEE Computer Society Press, 1999

[20] 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

[21] Holtzmann G. J, "The Model Checker SPIN" *IEEE Transactions on Software Engineering*, Vol 23, No 5, 1997

[22] Jackson, D, "Alloy: A lightweight Object Modelling Notation" available at http://sdg.lcs.mit.edu/ dnj/abstracts.html, July 1999

[23] Kurki-Suoni, R "Component and Interface Refinement in Closed-System Specifications" *Proceedings of FM'99*

[24] Luckham, D.C et al, "Specification and Analysis of System Architecture using Rapide", *IEEE Transactions on Software Engineering*, April 95

[25] Luckham, D.C and J. Vera, "An event-based architecture definition language", *IEEE Transactions on Software Engineering*, September 95

[26] Luckham, D.C, "Rapide: A language and toolset for simulation of distributed systems by partial orderings of events", see http://pavg.stanford.edu

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

[28] Magee, N. Dulay, S. Eisenbach and J. Kramer. "Specifying Distributed Software Architectures", *Proceedings of 5th European Software Engineering Conference (ESEC 95)*, Sitges, Spain, September 1995

[29] Magee J and Kramer J "Concurrency: State Models and Java Programs", Wiley, 1999

[30] Object Management Group "Common Object Request Broker: Architecture Specification" see http://www.omg.com

[31] Shaw M et al, "Abstractions for Software Architecture and Tools to Support Them", *IEEE Transactions on Software Engineering*, April 95

[32] Shaw, Mary and Garlan, David "Software Architecture - Perspectives on an emerging discipline". Prentice Hall, 1996

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

[34] 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

[35] Sun Microsystems, "Jini Software Simplifies Network Computing" available at www.sun.com/jini

[36] Wile, D, "AML: an Architecture Meta-Language", *Proceedings ASE 1999*, IEEE Computer Society Press, pp 183-190