Reasoning about Asynchronous Behaviour in Distributed Systems

Peter Henderson

Department of Electronics and Computer Science
University of Southampton
SO17 1BJ, UK
peter@ecs.soton.ac.uk
http://www.ecs.soton.ac.uk/~ph

Abstract

When a new component is added to an existing, distributed system, it has to co-operate with existing components in a way that doesn't interfere badly with the original system. Adding new components to an existing system is simplified if their communication is asynchronous. It allows for looser coupling. Unfortunately, the fact that the communication between components is asynchronous adds to the difficulty of reasoning about their behaviour. This paper gives an illustrative example of a simple distributed system with asynchronous behaviour and discusses how its behaviour can be described and reasoned about in terms of goals. This formalises what we believe to be contemporary engineering practice. Experimental support for reasoning, including animation, is particularly appropriate and practical in these circumstances, because the properties which we must reason about are emergent rather than compositional.

1. Background

In large scale, distributed systems of the type that global enterprises depend upon, business process evolution is a particular problem. If a new business activity requires new software support, this usually manifests itself as a set of new applications which must co-exist with a set of existing applications and work together with them successfully [13]. The software engineers responsible for designing, building and ultimately delivering the new applications must reason about the potential behaviour of the system as a whole and what will happen after the addition of the new applications.

In general, the behaviour of the system as a whole can be described in terms of properties to be maintained and goals to be achieved. An example of a property to be maintained is that, where old functionality is not being replaced or modified, it should continue to be available to the users as it was before the evolution. An example of a goal to be achieved would generally be more specific, based on the detailed functionality of the old or new applications. For example, if the global business was banking, and the new functionality was a new financial product to be sold over the internet, a goal of the evolved system could be that any customer who engages in purchasing that product should be able to complete that transaction promptly. You can imagine how that might be expressed more formally in a real case.

One way to characterise this type of problem, which enables us to discuss the nature of the difficulties which face the software designer, is to consider a system architecture which allows the free addition of new, collaborating applications. In [10] we discussed, informally, such an architecture and the rules which new applications would have to obey if they were to be able to co-exist within an already functioning system without disrupting it. Since then, with architectures such as MQ, .NET and J2EE, for Enterprise Application Integration and in particular integration of web-services, these problems have become more, rather than less, prominent.

2. The Nature of Enterprise Systems

One way of looking at large-scale, distributed systems is as an integration of co-operating applications [1, 6, 7, 8, 13, 17]. Figure 1 depicts a set of applications, including A and B, co-operating within a system S. We shall assume that the applications A and B communicate with each other, and with others, by asynchronous message passing. This is without loss of generality, since synchronous communication between applications, even with Remote Procedure Call (RPC) and Remote Method Invocation (RMI), is rare in enterprise systems. This is because, in these circumstances, the total communication will usually consist of a series of

calls and some time will elapse before that sequence is completed. In order to ensure that this sequence of synchronous calls is valid, the engineer has reasoned about the asynchronous effect of the sequence. This is not to deny the importance of transaction processing, which makes asynchronous behaviour look synchronous. In particular, distributed transaction processing is very important. But this importance is for behaviour which is "near" databases. We are addressing extended transactions over heterogeneous applications and long time periods. Transaction technologies, for us, happen within applications. They prevent applications from interfering with each other near databases.

The situation we are particularly concerned with is, what happens when a new application (C, say) joins an existing, running system? What can C assume about the other applications, some of which it must communicate with? Conversely, what can they assume about C, so that they are not damaged?

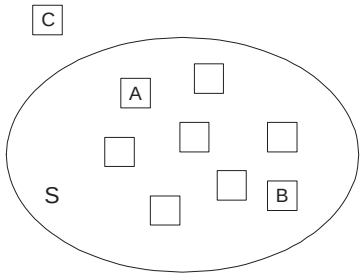

Figure 1. A system with a component waiting to join

These are important questions to answer if we are going to fully benefit from the plug-and-play opportunities which component-based systems offer. It is not sufficient that the components (in our case, applications) have plugs and sockets which fit each other. This is just syntax. It is also necessary that they have behaviours which fit, in that they work together, in particular new with old.

In a web-services based architecture, the usual objective is to publish a new application and have it provide a service to existing and future applications. Providing a service is relatively straightforward, in that the service provider can dictate constraints which the service user must obey if they are to avail themselves of the service. The service provider can protect itself from bad users and it can even provide services to existing users (applications) by publishing its service in the form they expect. So this gives us the mechanism, both to evolve a service and to keep faith with legacy applications which use it. But, how do we know that the enhanced behaviour will combine in a satisfactory way with existing applications?

This is the context in which our problem is cast. Previously [13], we have addressed this problem informally, in the context of enterprise scale components (Shops, Customers, Malls etc.). In this short paper we give an example of a much simpler system with many of the properties of large-scale, asynchronous distributed systems. The example is small enough that we can discuss more formally the type of problems that arise in predicting the behaviour of such systems under change. In this context, we can also discuss the contribution of animation to the process of design validation.

3. Asynchronous behaviour

Consider the applications shown in Figure 2. These are supposed to be part of a very simple (very abstract) trading system in which items are traded for cash. The participants (applications) can transmit items or cash to each other in messages. These messages take time to travel and hence reasoning about the behaviour of the system as a whole necessarily means that we need to reason about asynchronous behaviour. In the next Section, we will consider a more extensive example. Here we concentrate of defining a few concepts.

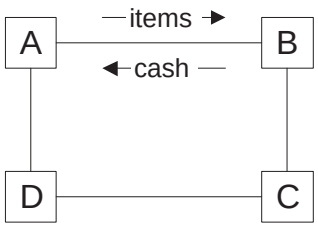

Figure 2. A system of trading components

First, let us look at a possible behaviour for a component such as A in Figure 2. Suppose it behaves as follows:

```
do forever
   receive cash from q;

do forever{
   receive item from q;
   send cash to q;
}

for each q in P
   do forever
     send item to q;
```

This informal notation needs some explanation. The `do forever` construct introduces a new thread running

in parallel with the thread which created it. As its expression suggests, the thread is an infinite loop. The above behaviour description introduces a number of threads. The first of these is able to receive messages of the form $cash$ from anyone. The receipt of a message binds the name of the sender to the parameter q, which this first thread just ignores. We are working at a level of abstraction which concerns only the message-passing behaviour of our applications.

The second thread receives an $item$ and sends a $cash$ message in reply, thus using the name of the sender selected from the incoming message. We assume that message passing is buffered and that as a consequence sends will seldom block. If we had been concerned that the send might block we could have created a new thread using `thread{send cash to q;}`, which spawns a new thread to execute the single statement in parallel with its creator, and then stops. Note that we assume an implicit `thread` prefixes each occurrence of `do forever` (so `do forever S` is equivalent to `thread{for(;;)S}` to use a C/Java idiom).

The third construct in our behaviour description uses a constant P which is the collection of other applications which this one can communicate with. It then sets up $|P|$ threads which will send arbitrary items to each other participant at arbitrary intervals. We could, of course, have coordinated the receipt of cash with the sending of items. We have chosen not to do this, because the behaviour we have given has some interesting consequences. We rather assume that the application keeps records of which items have been paid for. Consequently, each $cash$ message will contain an identification of which $item$ it refers to. At any point in time an application will have some items that it has sent, but which have not been paid for, and possibly one item it has received which it has not paid for. It is possible that payments will be received which refer to items which have not been sent. Reasoning about this eventuality should tell us that this cannot happen, if our applications and their supporting communications structure behaves correctly. Of course, such things do happen in practice when our system is failing.

Let us now look more carefully at the behaviour of our application, in the context of the system shown in Figure 2. The application A, executing the above program, assumes that all the others are executing the same, or consistent, programs. The application A has a goal. It is to send as many $item$s as it can and to receive as much $cash$ as it can. The fact that it trusts its customers to pay up is not an issue we discuss, it is part of the application's business process. Rather, we are interested in how we as engineers reason about the behaviour of this application in a context such as we have shown here. Since messages take time to travel, we are reasoning that cash will eventually arrive as a consequence of items sent. The application's goal is of the form *collect cash for item sent to q*. In fact, we have a set of goals, one for each outstanding item sent to q and many for each q in our set P of acquaintances. These are not the only goals. Another is to be honest and pay for everything we receive. Finally, there are goals which are simply to send (apparently unsolicited) items in the hope that others act like us and always, eventually, pay up. Of course, this is just an abstraction of a more reasonable concrete behaviour.

The reasoning which the engineer uses to design an application such as this, can be formulated as attempting to reach a set of goals. Knowing that the consequences of actions will be delayed by the time messages take to travel (and the time others take to get round to replying), the engineer reasons as follows: assuming others behave in ways which I have predicted then the consequences of the actions performed by my application will be to achieve one or more of the goals I have set. To try to validate the design decisions, the engineer will reason about special cases, such as one copy of the application talking to another copy of the same application. Does that behave appropriately? Another special case is where it is imagined that all participants in a scenario cease proactive behaviour and the whole system comes to a quiescent state, ridding itself of all undelivered, unresponded-to messages. Will the application have arrived at an appropriate state? For example, will all outstanding payments have been made? The answer for our application is affirmative, assuming every other participant sends cash reactively.

The interesting issue is that, while we reason about these special cases of behaviour, we never expect to see them in practice. In an asynchronous system such as this we will always have a large number of messages outstanding. From the viewpoint of a single application, it will always have goals that are not yet met.

Consider now what happens when an application, plug-compatible with the one above, but behaviourally different is added to the mix. What are the consequences for it? And what are the consequences for the existing, legacy applications? For example, the above application is both buyer and seller. It could be specialised to have only one of these behaviours. Would those specialisations work with each other and with the more general component. At this level of complexity it is straightforward to see that they would. There are many derivatives of this application which we could develop. Eventually, we get into a more complex situation. We get into the situation where behaviour is emergent rather than compositional. We get into a situation where the predicted behaviour is seen, and progress continues to be made towards the goals each application has set for itself, but where additional unpredicted behaviour is also seen. This additional behaviour emerges as a consequence of the interactions between applications. Once seen, we can usually

reason about its consequences in much the way that we reason about the consequences of our own decisions. The problem is seeing it in the first place. This is where modelling and animation can help. Before we discuss that, however, we need a more illustrative example.

4. A more illustrative example

Let us look at another very simple example of asynchronous behaviour. This example is supposed to have the characteristics of a set of enterprise applications, working together, but be simple enough that we can illustrate some of the problems that arise when we reason about changes we plan to make to such a system.

The example is a simple, self-organising collection of applications. We imagine that these applications are objects in two-space. Indeed, since this is just an illustrative example, all we are interested in is their location. We are going to organise that they keep sufficiently far apart from each other so as to avoid the possibility of collision. And we are going to organise that they keep sufficiently close together so as to occupy a restricted amount of space. We are going to insist that the applications communicate with each other asynchronously and that the only way that any application can know the location of any other is by receipt of a message, which will necessarily contain information which is a little out-of-date.

This is a very artificial example. It does however allow us to make our points about how to formulate behaviour in such systems and how to reason about that behaviour both locally and globally. And it does contain some of the essence of the behaviour of typical Enterprise Application Integration solutions.

In our simple example, the behaviour we will give each application is identical, and as follows

```
do forever
   for each q in P
      send myLocation to q;

do forever {
   receive qLocation from q;
   if (| myLocation - qLocation | < 100)
      moveAwayFrom qLocation; else
   if (| myLocation - qLocation | > 400)
      moveTowards qLocation;
}
```

This defines two threads, each running forever. The first thread sends the location of this application, repeatedly, to all other applications. We assume that P is the set of all applications other than this one. The second thread receives messages from each other participant (application), in indeterminate order, and makes a suitable move away from or toward the location of that application. The message passing is buffered, but the buffers may be of restricted length. We will discuss the effect of restricting the size of buffers in due course.

The two-space in which the applications are arranged has been artificially restricted. The size of the available space (400) and the size of the separation (100) are arbitrary, of course. These are sizes which have been used in some of the experiments reported on here. A little combinatorics, and a little geometry, suffices to prove that they allow for up to 19 applications to eventually find homeostasis. If 20 applications are set running with these parameters, not all of them will reach their goals.

This is exactly the kind of behaviour we observe in Enterprise Application Integration. That is why a problem such as this is relevant to the issue of reasoning about Enterprise Application Integration.

In Enterprise Application Integration we will have many different applications running and exchanging messages [2, 3, 9]. Each will have a set of goals which it is trying to achieve. Over time it will achieve some of these goals and acquire others. Should it ever reach a point where it has achieved all its goals, it will be quiescent, until another goal arrives or another message instantiates a generic goal and the application sets about achieving that. It is this formulation in terms of sets of goals which we believe is the way that engineers reason about the expected behaviour of the complex systems which constitute large-scale Enterprise Application Integration solutions. The simple example we have chosen allows us to illustrate this formulation in terms of goals, without having to encumber that description with e-Commerce domain details.

Suppose we have a number of applications running, all executing the algorithm given above. We can characterise their behaviour in terms of a set of goals to be achieved. For a given application p assume that P is the set of all applications other than p. Now, if q is in P, we will have the following two goals (for p with respect to q)

$$G1 : |pLocation - qLocation| \geq 100$$
$$G2 : |pLocation - qLocation| \leq 400$$

where $pLocation$ and $qLocation$ are the vector positions of the applications p and q respectively. In fact there will be $n-1$ pairs of such goals for p, if we assume there are n applications altogether in our scenario.

It is important to think of these $2n-2$ goals as separate. Hence we refer to a set of goals to be achieved by an application. The reason is that we are going to discuss situations where our system (the set of applications) is making acceptible progress (i.e. doing what it was designed to do) but where the conjunction of these goals is never achieved. Note that a goal is not an invariant. Usually a goal is not true, we are striving to make it true. It may be that an appli-

cation has properties which it wishes to maintain as invariant, but these are not its goals in the sense in which we are using the term here.

Using the combinatorics and geometry referred to earlier, we know that at most 19 applications can be supported if all are to reach the quiescent state where they never move again. That is, where all have reached homeostasis. If 20 or more are running, they will progress to a state where some (perhaps zero) will reach that state, but where not all will.

How do we know this? Actually, it is quite difficult to prove that 19 applications, all executing the above program, will eventually settle. This proof will have to be based on some notion of fairness. It is equally difficult to predict the eventual state of a system with 20 or more applications. Again, fairness would play an important role. It is actually quite difficult to describe the state which 20 applications might reach, since we know they will not all remain stationary. In this simple example, we resort to stating that they might perhaps settle into a restricted region of two-space. Experimental evidence suggests this is what happens. This is a particularly simple example of *nearly-met* goals. Notwithstanding these difficulties, the program we have written for each application is in some sense "obvious" and one we would anticipate might achieve some, if not all, of the goals of each application. From experiment, we conjecture that the 20 applications which never settle always arrange themselves in a recognisable global state, regardless of buffering. The experiments we have performed can be observed, with a suitably configured browser, at the URL given in [24].

This is where our formulation in terms of sets of goals comes in. Consider the components shown in Figure 1. Suppose these components are our applications and that they are all executing the program given above. Consider the situation between A and B, from A's point-of-view. At some early stage the goals $G1$ and $G2$ for A (with respect to B) may be not-met, because the applications are too close or too distant. At some later stage, they may be met, because the behaviours of A and B both move towards achieving this. Later still, because of the existence of other applications, A may find itself in a state where again one or more of its goals are not-met. Thus we can see the way that behaviour is described in terms of sets of goals. A has a set of goals, $2n - 2$ of them in this simple example. At any point in time it has satisfied a subset of them. Occasionally all of them. It makes progress by trying to satisfy goals in its current set. Its ultimate state is not quiescent. Large, enterprise-scale systems exhibit this type of behaviour. In practice, we are able to predict their progress accurately by describing progress in terms of goals being met.

This formulation also allows us to incorporate the arrival of new applications, necessary if we are to reason about software evolution. In Figure 1 assume that C is a new copy of our simple application. Adding it to S is incorporated in our behaviour model, because our formulation considers behaviour to be localised. That C modifies the behaviour of others is modelled by their adoption of additional goals, although we will need to explicitly modify their behaviour in some circumstances. This is an issue we return to later.

The application we have described is apparently "stateless". If we consider that it does not remember which items it has sent, then it is indeed stateless. If we consider that it saves its state on stable storage, then we can treat it as stateless. Consequently an application is partially robust against failure. It can restart and continue in a sensible manner. To make it totally robust against failure, in particular against the receipt of inconsistent information, we would need to arrange that a symmetric, end-to-end agreement was implemented. This is a topic we note here, but which we shall not address.

Observe that the program given above is not the only program we could have chosen. Indeed, it is far from the more sophisticated programs used to achieve goals of this sort in practice. In particular, the goals themselves are not held explicitly by the application, although in this case they are simple reformulation of the two actions which the application takes. However, even this simple program has some interesting idiosyncracies, which its designers must be able to reason about if they are to deploy this application safely.

The first idiosyncracy is how the application's behaviour is dependent upon the buffering in the message system. Suppose that messages, containing location information, are quite considerably delayed. By the time A acts upon a message from B, it (B that is) may well have moved. A will then move towards or away from where B was, rather than where it is now. Will this have significant effect upon global behaviour? More importantly, will this have significant effect upon A's local perception? Should the designers worry about this and ensure, to the extent that they can, that the messages are not delayed unduly? Or that delayed messages are not acted upon?

It is here that experimental evaluation comes into its own. It is here that animation can serve to support our reasoning, as follows. We collect experimental evidence of certain behaviour, sufficient to construct a conjecture which we might test and ideally establish by some formal means.

For example, in one experiment we ran, using a simple graphical animation package, a certain configuration of applications nearly-settles when the asynchronous messages are not delayed. It is not obvious a priori what will happen if messages are delayed, especially if they are delayed by a substantial amount. Experimental evidence suggests that for quite significant delays, the configuration continues to nearly-settle. This is not proof that it will always happen but it does lead to the obvious observation that to nearly-settle is stable. That is, once achieved, it will remain. If

the applications have nearly-settled, it doesn't matter how delayed the messages are because the information they contain will be nearly-accurate. The global observation is that, information about where objects were is a good approximation to where they may be now, especially since settling down is an emergent behaviour of the system in which our applications find themselves.

Settling and nearly-settling are stable, emergent behaviours. A system, having settled, will remain settled. A system, having nearly-settled, will remain nearly-settled.

A second experiment which we performed was as follows. Suppose the new application C has a different behaviour from the existing applications (A, B, etc). How will this affect the behaviour of those older applications. In particular, suppose we give C the following behaviour.

```
do forever
   for each q in P
      send myLocation to q;

delta=0;
do forever {
   receive qLocation from q;
   if (agitated)delta=delta+1;
   if (| myLocation - qLocation |
                        < 100 - delta)
      moveAwayFrom qLocation; else
   if (| myLocation - qLocation | > 400)
      moveTowards qLocation;
}
```

Here, we have added the ability for new applications to cooperate by being willing to be closer to others by an amount which they increase if they are agitated. We won't give the details of "agitated" here. Suffice it to say that an application is agitated if it has moved back and forth over a small region for some time. How will this new application behave? Will a new application of this sort, added to a settled situation, fit in? The interesting case, given the parameters we have used here, is if the new application is the 20th. The 19 original applications would settle, as we claimed earlier. Would the new application's tolerance create enough room for it to settle? After some experimentation it is eventually obvious that it will not. The new application's tolerance of proximity is not matched by the old application. So whenever the new one moves close to an old one, the old one moves away. While a scenario with all new applications does settle, no mixture of old and new does.

The reason is that the goals of the old application remain as they were. To modify their behaviour we have to modify their goals. We have done these experiments too, allowing old applications to download new behaviour shown above. In this simple scenario, the old applications were allowed to download the new behaviour. Eventually, in every experiment we ran, every old application eventually took on the new behaviour. This experiment can be observed at [24].

5. Practical reasoning

We have argued that complex distributed systems, of the sort that constitute Enterprise Application Integration solutions, are reasoned about by their designers in terms of local goals. In practice, individual applications are conceived of as always having an outstanding set of goals which they are trying to achieve. At any point in time one or more of these goals may be achieved. New goals arrive, either as a consequence of new information instantiating generic goals, or as new business objectives. These new goals may require new behaviour. Eventually every goal may be achieved, yet we may never have the situation where an application reaches a quiescent state where all goals have been discharged. Nevertheless, we succeed in building such applications and we do this by using practical means of reasoning about them.

Our reasoning can benefit from mechanical support. Model checking [5, 15, 18] has proven its practicality in these situations. It has been successfully applied at an enterprise level [14, 20]. The advantages of model checking are that behaviour can be observed, at first by animation, but then more comprehensively by setting up experiments which test safety, liveness and progress properties. This experimentation leads to conjectures which we reason about outside the support of the model checker. The experiments lead to the development of arguments which we use to convince ourselves, and others, that our solution is "correct".

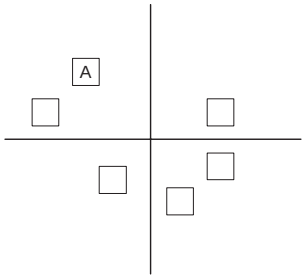

Figure 3. Six applications in four adjacent cells

Consider the arrangement of applications shown in Figure 3. This is an abstraction from the illustrative example used in the previous Section. This time we have six applications A, each trying to avoid the others. Each has a goal to be in a quadrant on its own. Clearly this cannot be achieved since there are only four quadrants. However, when this conjecture was given to a model checker [14], it quickly arrived at a contradiction. In fact, it found that the above arrangement of applications was such that each believed its goals were satisfied. The reason is because of the way

that applications only come to know of the co-occupancy of cells by messages. While these messages are in transit, it is possible that some applications (maybe even all applications) believe they have achieved their goals, when in fact that situation is just about to change. Formulation of an experiment of this sort, for a model checker, confirms our conjecture that behaviour is clearly expressed in terms of goals. A common form of progress property to check with a model checker is of the form *always, eventually G* which is a temporal way of saying that, no matter what state we are in, eventually we will be in a state when G is satisfied. But, of course, that doesn't mean it will remain true.

Experiments of this sort notwithstanding, there are limits to how far we can achieve the insight we need, using just the kind of experimentation which model checkers support, especially when behaviour is emergent, rather than being the direct consequence of the composition of the behaviours of its constituents. It is not straightforward to observe this emergent behaviour from the kinds of experiments which model checkers encourage. Here there is a role for animation of the type which allows clear visualisation of the emergent behaviour. The LTSA model checker [19] now supports some very powerful visual animation, so this may well be a trend in model checking. What we have used to perform the experiments reported here is a rather more ad hoc graphical animation package.

Of course, the example we have chosen lends itself to that sort of visualisation. We can place a graphical rendition of each application on a screen and see them moving, settling, being agitated etc. It is not immediately obvious how one would visualise the more realistic enterprise applications, although [23] is full of interesting ideas in this direction.

Previously, we have looked at modelling systems from a component-based point-of-view [11, 12, 14], without concerning ourselves with the distinction between synchronous and asynchronous behaviour. Enterprise-scale systems however require this. The formal modelling of asynchronous and distributed behaviour in the context of evolutionary change is particularly challenging [21]. It leads to emergent properties, which are difficult to predict without experimental evidence [17]. Once seen, however, these properties can be cast in the form of striving towards local goals and the system behaviour can be seen as the amalgamation of these local behaviours. For us, animation has formed the basis of the usual process of conjectures and refutations which lead to a detailed understanding and an ability to reason formally about complex behaviour.

6. Conclusions

In this short paper we have shown how the behaviour of individual components in a complex distributed set of applications can be described in terms of the set of goals each is to achieve. We gave an example of a simple distributed system with asynchronous behaviour and discussed how its behaviour can be described and reasoned about, formalising what we believe to be contemporary engineering practice. In particular, we demonstrated that this behaviour can be formulated in terms of local goals. We argued that experimental support for reasoning, including animation, is particularly appropriate and practical in these circumstances, because the properties which we reason about are emergent rather than compositional. We showed that our simple example incorporates many of the properties of large-scale Enterprise Application Integration problems.

References

[1] Martin Bichler, Arie Segev and J. Leon Zhao. Component-based E-Commerce: Assessment of Current Practices and Future Directions. *ACM SIGMOD Record* 27(4): 7-14, 1998

[2] Ken Burgett. JMS and EJBs: The inbound message conundrum. *http://www7b.boulder.ibm.com/wsdd/library/*, 2001

[3] Antonion Carzaniga, David S. Rosenblum and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems (TOCS)*, Volume 19, Issue 3, 2001

[4] L. Chung and N. Subramanian, Architecture-Based Semantic Evolution: A Study of Remotely Controlled Embedded Systems. *Proc., IEEE Int. Conf. on Software Maintenance (ICSM'01)*, 2001

[5] E.M. Clarke, Orna Grumberg and Doron Peled. *Model Checking*, The MIT Press, 2000

[6] Darach Ennis. CORBA and XML Integration in Enterprise Systems. *IONA Technologies Inc.*, 2000

[7] Faramarz Farhoodi and Peter Fingar. Developing Enterprise Systems with Intelligent Agent Technology . *Distributed Object Computing*, Object Management Group, 1997

[8] Peter Fingar, Harsha Kumar and Tarun Sharma. *Enterprise E-Commerce*. Meghan-Kiffer Press; ISBN: 0929652118; 1st edition, 2000

[9] Hassan Gomaa, Daniel A. Menasc, Michael E. Shin. Reusable component interconnection patterns for distributed software architectures . *Proceedings of Symposium on Software Reusability (SSR '01)*, ACM Software Engineering Notes, Volume 26, No 3, 2001

[10] Peter Henderson. Laws for Dynamic Systems. *Proceedings of International Conference on Software Reuse (ICSR 98)*, p.330-336, IEEE Computer Society Press, 1998

[11] Peter Henderson and Robert John Walters. Modelling Component-based Systems as an aid to Design Validation. *14th IEEE International Conference on Automated Software Engineering (ASE)* p.303-6., 1999

[12] Peter Henderson and Robert John Walters. Behavioural Analysis of Component-Based Systems. *Information and Software Technology*, 43 p.161-169. 2001

[13] Peter Henderson. Asset Mapping - developing inter-enterprise solutions from legacy components. in *Systems Engineering for Business Process Change - New Directions*, Springer-Verlag UK, pp 1-12, 2002 see http://www.ecs.soton.ac.uk/~ph/papers

[14] Peter Henderson. Modelling Architectures for Dynamic Systems. in *Programming Methodology*, Edited by A. McIver, C. Morgan, Springer Verlag Monographs in Computer Science, 2002, see http://www.ecs.soton.ac.uk/~ph/papers

[15] Gerard J Holtzman. The model checker SPIN. *IEEE Transactions on Software Engineering*, Vol 23, No 5, 1997

[16] Jens H. Jahnke. Engineering component-based net-centric systems for embedded applications. *Proceedings of the 8th European Software Engineering Conference*, ACM Software Engineering Notes, Volume 26, Issue 5, 2001

[17] Havard D. Jorgensen and Steinar Carlsen. Emergent Workflow: Planning and Performance of Process Instances. *Workflow Management '99*, 1999, see http://www.wi.uni-muenster.de/is/Tagung/Workflow99/

[18] Jeff Magee and Jeff Kramer. *Concurrency : State Models & Java Programs*. John Wiley and Sons Ltd (1999)

[19] Jeff Magee, Nat Pryce, Dimitra Giannakopoulou and Jeff Kramer. Graphical Animation of Behaviour Models. see http://www.doc.ic.ac.uk/~jnm/book/ltsa-v2/animation_paper.pdf, 2000

[20] Shin Nakajima and Tetsuo Tamai. Behavioural analysis of the enterprise JavaBeans component architecture. *Proceedings of the 8th international SPIN workshop on Model checking of software*, 2001

[21] Gian Pietro Picco, Gruia-Catalin Roman and Peter J. McCann. Reasoning about code mobility with mobile UNITY. *ACM Transactions on Software Engineering and Methodology*, Volume 10, No 3, (2001)

[22] R. Sesseler. Building agents for service provisioning out of components. *Proceedings of the Fifth International Conference on Autonomous Agents*, 2001

[23] Robert Spence. *Information Visualization*. Addison Wesley, 2001

[24] See http://www.ecs.soton.ac.uk/~ph/reasoning