Behavioural Analysis of Component-Based Systems

Peter Henderson (P.Henderson@ecs.soton.ac.uk)
and
Robert Walters (R.J.Walters@ecs.soton.ac.uk)

Declarative Systems and Software Engineering Research Group
Department of Electronics and Computer Science
University of Southampton
Southampton, UK.
SO17 1BJ

16 December 1999

1. Abstract

Software Engineers continue to search for efficient ways to build high quality systems. Two contrasting techniques that promise to help with the effective construction of high quality systems are the use of formal models during design and the use of components during development. In this paper, we take the position that these techniques work well together.

Hardware Engineers have shown that building systems from components has brought enormous benefits. Using components permits hardware engineers to consider systems at an abstract level, making it possible for them to build and reason about systems that would otherwise be too large and complex to understand. It also enables them to make effective reuse of existing designs. It seems reasonable to expect that using components in software development will also bring advantages.

Formal methods provide a means to reason about a program (or system) enabling the creation of programs which can be proved to meet their specifications. However, the size of real systems makes these methods impractical for any but the simplest of structures - constructing a complete formal specification for a commercial system is a daunting task.

Using formal methods for the whole of a large commercial system is not practical, but some of the advantages of using them can be obtained where a system is to be built from communicating components, by building and evaluating a formal model of the system. We describe how a model of a system to be implemented using COM might be constructed using a particular modelling tool, RolEnact. We discuss the extent to which validation of the model contributes to the validity of the eventual solution.

2. Introduction

It seems obvious that the widespread adoption of formal methods in software development would lead to a major improvement in the reliability and general quality of software systems. However, the effort required to use formal methods, and hence the cost, increases rapidly with the size of the system. As a consequence, despite the advantages their use has not spread far beyond the development of safety critical systems [22].

Where the use of formal methods throughout the development of a system is prohibitively expensive, the developers should not simply dismiss these techniques. There may still be opportunities to use them which will bring benefits to the development process. For example, there may be a sub-system where their use is appropriate such as encryption of messages in a communications system where security is important .

However, we believe that they can also be used to evaluate more than small sub-systems if

developers construct suitable models of a proposed system. These models can be evaluated using formal techniques like model checking [7-9, 13, 24, 25] as well as less formal techniques such as manual inspection of an executable version of the model [1-3, 11, 17, 19, 20, 29, 33]. Although this can never amount to proof of the "correctness" of a system, it can give some measure of confidence in the design of the system. The models can also be used as a vehicle for explaining aspects of a design (or a problem) to the client and members of the development team. Once the models are completed, they can be used to form a template for the generation of the components that are to form the delivered system. If the modelling technique used is suitable, the models should be quick and easy to create and use.

The major advantage of building models is that the size (and hence the cost and complexity) of the models can be managed by using appropriate abstractions from the real system.

3. The Three Layer Model and Components

Early business system typically ran on a mainframe computer installed in some central location. Users of the system communicated with it by sending and receiving paper forms, or using dumb terminals. No part of the application or processing was done other than on the mainframe.

In recent years, we have seen the decentralisation of computing from large mainframe computers onto increasingly powerful desktop machines and the arrival of the "client-server" architecture in which the role of centralised computing facilities has been reduced to little more than data storage. As this revolution has progressed, more and more of the processing carried out by the system has migrated to the remote "client" machines leading to a situation where the role of the central machine has diminished to just providing data access and storage services. With this arrangement, the role of the application on the "client" machine has become much more than that of providing an interface to the user of the system. However, it is becoming recognised that the software running on these "client" systems is doing much more than providing the interface to the user and that this function belongs in a new layer of the system which connects the user interface to the storage objects.

Commonly, a solution which is to be realised using a client-server architecture is described using a three-layer model (see Figure 1). In this three layer model, the two layers of the "client-server" architecture become the data storage and user interface layers and these are connected by a new layer which is concerned with "business rules". This layer provides the connection between the data storage of the system and the user interface and is the layer which contains the information and algorithms about how the system is to perform its various tasks and transactions. Keeping the business rules out of the data storage layer and out of the user interface layer is one of the key objectives of a flexible client-server solution.

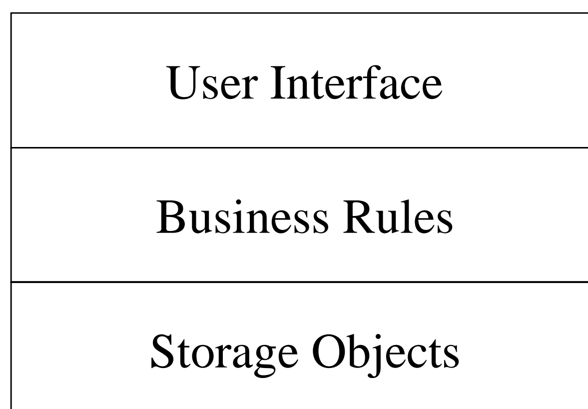

Figure 1: The Three Layer Model

The Storage layer of a three layer system provides back office systems like database management systems and permanent storage of data. In many systems this will be provided by a bought-in database management system. The user interface layer is concerned only with interactions with users of the system. They are intended to present information to and elicit information from the users. Their operation is independent of the underlying task of the system. This means that they can be light weight applications that can be changed easily and cheaply as the demands of users change.

The business rules layer joins the other two and contains information about the purpose and operation of the system: for example consider an interaction with the system in which a user views and updates or corrects some data stored in the system. This data is stored in the data storage layer of the system. The user interface might present a completed form to the user and have a button to update records reflecting any changes or corrections. Understanding the relationship between the fields on the form and the data structures in the data storage layer and how properly to manipulate them is the responsibility of the business rules layer of the system.

Extending the "client-server" architecture, in the three layer model (or multi-layer model) view of a system there is no expectation that individual layers will correspond to identifiable units of hardware. For example, an organisation might choose a system which uses dumb terminals for some reason leading to the user interface software running on the central machine, or the storage objects might be implemented using a distributed database application. However, the most likely pattern would be for the user interface applications to run on a collection of machines distributed around the organisation using the system and the storage objects are most likely to be located on one or more "server" machines in a central location. The software associated with the business rules layer might be located on a machine (or machines) dedicated to the purpose, located on the machines providing the storage, or distributed about the system on the "client" machines. In some very small systems, it may be appropriate for the business rules to be incorporated into the User interface or the storage objects (leading to a "client-server" type of system). The way in which contemporary solutions are implemented is to modularise the software within each layer into one or more components. This separates the concerns of deciding how these components should communicate to deliver the required results from the concerns of placing the components on the workstations or on the servers.

Architectures have emerged which help developers to create these multi-layered systems and include CORBA [30], Enterprise JavaBeans (EJB) [39] and COM/DCOM [4, 12, 37, 38]. These architectures provide the infrastructure for systems to be constructed from collections of communicating components which may be distributed around a network of machines. Building systems from components in this way provides a mechanism for the extensive reuse of existing code and for the designers to consider systems at a more abstract level as they do not need to be concerned with low level details of component interactions. The code to be used can be existing code written as part of previous projects or bought-in from outside suppliers. There may still be some need for new code, but this can be specified in the form of components that, once written will be available for use in future systems. The study of a system's architecture has become of considerable importance recently [2, 11, 18, 35, 36] and capturing its behaviour in models has highlighted the significance of design validation [26-28]. For component-based architectures, we will show that models built using RolEnact are well-suited to design validation.

4. Validating systems built from Components

The validation of a design for a system to be built from components differs from that where the system is to be built using more traditional methods. The reason for this is that responsibility for the correct operation of the components rests with their suppliers. The development of the system becomes assembling existing components into a coherent system including ensuring that where necessary, the system can cope with any errant behaviour of the components.

We observe that in hardware development, where the use of component technologies is much more widespread than software, extensive use is made of models and simulation of proposed systems [6, 16, 23]. It seems reasonable to hope that these techniques will bring similar benefits to software systems being built from components. Hardware designers have a choice of mature tools for modelling, evaluating and simulating components based systems but, the characteristics of hardware and software systems are so different that these are not suitable for software development and equivalent software targeted tools are not yet available.

The software designer attempting to validate a design for a component based system would like to have a tool which would enable the proposed design to be formally evaluated leading to proof that the eventual system will meet its specification once completed. However, even generating the specification of a large software system is a daunting task, so attempting to generate proofs for these systems is an unrealistic goal unless they are of manageable size and/or the cost is justified because they are considered to be "safety critical". Instead of working with the complete system, we propose that the developers should work with formal models of proposed systems.

The advantage of using a formal modelling technique is that such models lend themselves to a variety of analysis techniques such as execution and model checking. Such techniques cannot be used easily or effectively on an informal model. At the same time, if suitable abstractions of the system are used, the size and complexity of the model can be kept under control.

During the construction and analysis of the model, the developer will acquire a better understanding of the system and any errors and bugs discovered can be corrected with a minimum of cost and delay to the project. Also, since building these models does not require major investments of time and effort from the development team, if a proposed solution appears unsatisfactory, models of alternative solutions can be constructed and evaluated and the best solution adopted.

Clearly a system which has not been developed using formal methods can never be "proved" to be correct, but if it is built from a model which has been subjected to rigorous analysis, this can give some measure of confidence in the correctness of the design. Appropriate abstractions from the real problem and suitable tools should enable the developer to build and analyse models with a minimum of effort.

Finding suitable abstractions of a problem is not easy and will vary form system to system, but the features required of the modelling tool to be used for this work are more easily described. The tool must:

- Produce models that are easy for all potential users to understand, including non-technical staff
- Enable developers to generate models quickly and easily
- Provide simple mechanisms for models to be executed and analysed
- Be accessible to novice users

In section 6 we will describe a modelling tool, RolEnact, based on a finite-state modelling paradigm. We believe RolEnact addresses all these issues. In particular, we believe that the level of abstraction which the designer is asked to use is appropriate to the validation of designs which are going to be realised using communicating components. Before introducing RolEnact, we will briefly discuss this choice of level of abstraction.

5. Structure and Behaviour of Components

The tools presently available to software developers to assist with system design vary from simple notation and diagramming techniques to comprehensive development environments used by the formal methods community. Many of these tools, for example for UML [10], lack the formal underpinning required to enable their models to be executed or verified in other ways. This is not a problem for tools and techniques from formal methods, but instead they demand a high level of detail and are inaccessible to all but the most skilled of users.

When we design a large software system we concern ourselves with its structure (which parts is it to be made of) and its behaviour (how will these parts collaborate in order to realise our requirements). When building a system from components, we have the same concerns. Deciding how the components will be plugged together is the structural description. Deciding how they will each use the services of the others to realise our requirements is the behavioural description.

Most development environments for components are able to validate a high degree of structural integrity between components by checking types across the interfaces at system-build time. The same environments do not usually support behavioural validation, except by allowing the addition of probes to the implementation which allows it to be diagnosed as it runs. Such tools are only available very late in the development process, when much, if not all of the code is available.

But there are well-developed techniques for giving suitably abstract descriptions of intended behaviour [3, 5, 14, 15]. Most of these are based on some variation of the finite-state machine (or state-transition chart), wherein components are viewed as having state which undergoes change as a consequence of transitions. By choosing sufficiently abstract states to model, transitions can be both simple to describe and yet adequately detailed for detecting erroneous behaviour.

One such notation for behaviour description which is well suited to the modelling of components is Role Activity Diagrams (RADS) [31, 32]. Designed originally for Business Process Modelling, they fit the task of modelling components which realise business rules very well, as we shall show. Although RADS were based originally on Petri-Nets, they are just a variation on the traditional state-transition

chart. But the variation is very important. Concurrent behaviour is modelled by giving a finite-state model for each Role and by allowing Roles to synchronise by the elegant device of having them share transitions (a device borrowed from Petri Nets, but also found in CSP [21]).

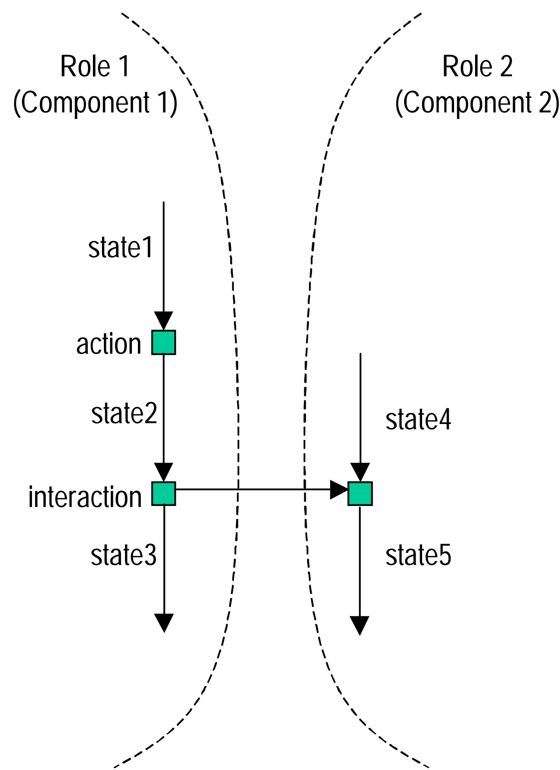

Figure 2: Syntax of a RAD

Figure 2 shows the syntax of a RAD. The unusual distinction from traditional state-transition charts is that in a RAD a state is shown as a line and transition as a small box. So in Figure 2 we see a transition labelled *action* which goes from *state1* to *state2*, and another labelled *interaction* which goes from *state2* to *state3*. Actually, the *interaction* is rather more complex, as its name suggests. It involves behaviour in two Roles. For the *interaction* to be performed *Role1* must be in *state2* and *Role2* must be in *state4*. When that is the case, *interaction* may occur, in which case both Roles change state, *Role1* going to *state3* and *Role2* going to *state5*.

In a RAD, each Role can have many concurrent threads of activity, and behaviour can also make conditional branches according to the results of internal actions or external interactions. We shall see examples of this later.

6. RolEnact

RolEnact [34] is a modelling tool which is based on RADS and which offers an approach to behavioural analysis for large component based systems. RolEnact models can be written as text or developed using a simple graphical interface which is able to display the model as a RAD-like diagram. The user can also execute the model in order to analyse its behaviour. In this section we describe RolEnact and in the next we illustrate how it can be used to analyse component-based designs.

A RolEnact model consists of a collection of “Roles” each of which has state and a number of “events” in which it will take part. When an event occurs, each of the Roles involved changes from a specified “Before” state to a specified “After” state. RolEnact formally addresses two issues that are rather informal in the definition of RADS. Firstly it addresses the issue of creating Roles. Secondly, it addresses the issue of how one Role knows of the existence of another.

In RolEnact, there are four types of event:

Action: A unilateral change of state by a single Role.

Interaction: An event where two Roles that know of each other interact resulting in both changing state.

Create: Creates a new Role of a specified type in an initial state.

Selection: An interaction between *any* two Roles of the required type whose states satisfy the "before" conditions of the event.

Creation and Selection events have the side effect that the Roles concerned create and store references to each other (enabling them to interact later).

Using a model of a proposed system, a modeller is able to step through anticipated sequences of events and experiment to see how the system behaves in different scenarios.

For definiteness, let us briefly, but formally, define the four types of event in RolEnact. We do this by assuming each Role is an object that can have attributes. In particular, every Role has the attribute *Role.state* which holds an atom representing its current state. Other attributes will be used to allow one Role to refer to another.

An Action is defined as follows:

```
Action(RoleName, BeforeState, AfterState) ≡
   If RoleName.state=BeforeState then RoleName.state:=AfterState
```

When executed, the Action, which is specific to a particular Role called *RoleName*, checks that the Role is in the state *BeforeState*. If it is, then the Role changes into the state *AfterState*. This models a simple internal event of the Role.

An Interaction is rather more complex. Its definition is:

```
Interaction(RoleName1, BeforeState1, AfterState1, RoleName2, BeforeState2,
AfterState2) ≡
   If RoleName1.state=BeforeState1 and
        RoleName1. RoleName2.state=BeforeState2 then
                {RoleName1.state:=AfterState1;
                 RoleName1.RoleName2.state:=AfterState2}
```

This time two Roles are involved. The first Role, *RoleName1*, is the Role invoking the interaction. It knows the identity of another Role, which it refers to by the attribute *RoleName2*. That is, the designation of the second Role is relative to the first: *RoleName1.RoleName2*. The behaviour is that both Roles are determined to be in appropriate states (*BeforeState1* and *BeforeState2*, respectively). They are then changed into the appropriate states states (*AfterState1* and *AfterState2*, respectively).

Create is one way in which one Role can have a reference to another. Its definition is:

```
Create(RoleName, BeforeState, AfterState, NewRoleName) ≡
   If RoleName.state=BeforeState then
                {RoleName.state:=AfterState;
                 RoleName.NewRoleName:=new NewRoleName;
                 RoleName.NewRoleName.RoleName:=RoleName}
```

Like a simple Action, Create is only effective if the Role which invokes it (*RoleName*) is in the appropriate state (*BeforeState*). In that case, it changes state (to *AfterState*), creates a new Role of type *NewRoleName* and establishes a reference to that Role by assigning the new Role to an appropriate attribute. It also records the reverse reference (from *NewRoleName* to *RoleName*), again using an appropriate attribute.

The other way in which one Role can have a reference to another is to use Select, defined as follows:

```
Select(RoleName, BeforeState, AfterState, SelectedRoleName,
        SelectedRoleNameBeforeState, SelectedRoleNameAfterState) ≡
   If RoleName.state=BeforeState and
        exists SelectedRoleName .
              SelectedRoleName.state=SelectedRoleNameBeforeState then
                    {RoleName.state:=AfterState;
                     SelectedRoleName.state:=SelectedRoleNameAfterState;
                     RoleName.SelectedRoleName:=SelectedRoleName;
                     RoleName.SelectedRoleName.RoleName:=RoleName }
```

This looks rather complex, but its behaviour is straightforward. Select is only effective if the Role which invokes it is in the right state (*BeforeState*) and there exists another Role (*SelectedRoleName*) which is also in an approprite state (*SelectedRoleNameBeforeState*). In that case, four things happen. The invoking Role changes state (to *AfterState*). The selected Role changes state (to *SelectedRoleNameaftereState*). A reference to the selected Role is established in the invoking Role. And a reference to the invoking Role is established in the selected Role

That is all there is to the behavioural definition of RolEnact. The RolEnact tool allows for models to be captured using a textual editor or a graphical editor for RADS (see Figure 7). It also allows models to be executed where the animation creates one window for each active Role. The modeller then interacts with the model by selecting a Role and invoking one of the actions displayed in the window for that Role. In this way the modeller can step through a scenario of the behaviour of the model. Again, we shall see an example of this later.

7. Mapping RolEnact to COM

Building and analysing (by execution or other means) a model of a proposed system gives a designer confidence in the proposed design. This may be useful in itself, but the greatest benefits from building the model will be gained if features of the model can be matched to their counterparts in the implementation. To do this, the modelling method needs to have a correspondence with the architecture to be used in the implementation.

As an example, consider how the features of a RolEnact model might map onto features of a COM implementation of a system. Figure 3 shows a comparison of the four types of event in RolEnact with how they correspond to the features of a COM based implementation of a modelled system.

Name	Effect in RolEnact	Effect in COM application
Action	Unilateral change of state	Internal operation of a COM object
Interaction	Communicate with a known instance of a Role	Send an event to/make a procedure call to a known instance of a COM object
Creation	Make a new instance of a Role	Make a new connection to a COM object
Selection	Locate and communicate with a selected instance of Role	Find and send an event to/make a procedure call on any existing COM object of the correct type

Figure 3: Comparison of Events of RolEnact and COM

COM is Microsoft's Component Object Model. It is the architecture on which technologies such as AtciveX and OLE are based. It is the architecture which has been used to realise many of Microsoft's current products, in particular Microsoft Office. Components in COM are often referred to a Objects, because the interfaces which they supply are like the interfaces to Objects (ie methods and events). But it is better to think of COM components as Servers, in that they are usually designed to supply a service (available at the interface). As such, they usually implement a whole Object Space within them, retaining responsibility for allocating and manipulating Objects of various types. A good example is the Excel spreadsheet application, which is a COM Component. It gives its user access to many different types of Object, such as WorkSheet, Cell, Formula etc. These Objects are created within the COM

Component and manipulated within it by the Component's user invoking actions at the interface of the Component.

Modelling such behaviour is ideally suited to finite-state techniques, such as RolEnact. Indeed, describing the behaviour of a component as a Role is exactly right. For a particular application, the general behaviour of the Component will not be what is required for a model. A suitable level of abstraction for the model will be more application-specific. For example, Excel may be used in a particular application, behind the scenes, to keep track of some complex financial calculations. The actions we wish to perform on the COM Component will be very specific, periodically sending data or requesting calculations. The Component is fulfilling a specific Role in the application.

8. A RolEnact Model for a Component-Based Design

We will illustrate this relationship between RolEnact and COM using a simple example. This example is of a banking system. The requirements of the system are that a number of counters are to be able to perform transactions to accounts and retrieve a balance for each account. There is also a requirement for a summary of the activity of the system. For the purpose of the example, only the user-interface and the business rules layers are considered. The eventual COM implementation of the design is shown in Figure 4. The system comprises three types of component: Counters where transactions and enquiries can be made, an Account Subsystem which is responsible for maintaining information about accounts and a Management Subsystem which provides summary information about the state of the system. It is accepted that the system may not be able to operate in the event of failure of an accounting component, but it is expected to be to continue to operate as counters open and close and in the absence of the management summary application. In a full sized system, there would be additional interactions between the Account Subsystem object and storage objects.

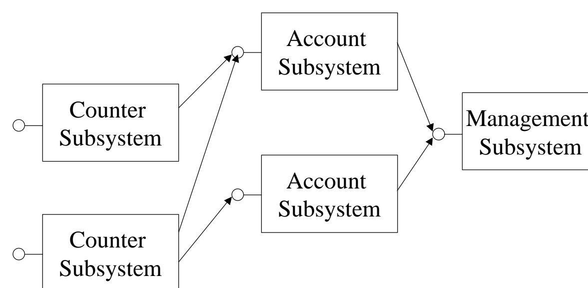

Figure 4: Banking Components

Figure 4 shows the structure of the proposed system. Some types of error in the planning of the system can be identified using the information contained in this diagram. For example, it is immediately obvious that the Management Subsystem must offer an interface to which Account Subsystems are able to connect and that Account Systems offer an appropriate interface for use by the Counter Subsystems. In some instances, it will not be possible to connect components in the desired configuration. This may be due to incompatibilities between the particular components concerned, but might also imply that there is no meaningful manner in which the components can interact. However, the presence of suitable interfaces to connect components does not imply that the completed system makes sense: there is a need to examine the behaviour of the components and their interactions. Like the considerations of the block diagram showing the connections between the components, this examination of behaviour need only consider the essential features of the interactions. The details of the these interactions can be added later.

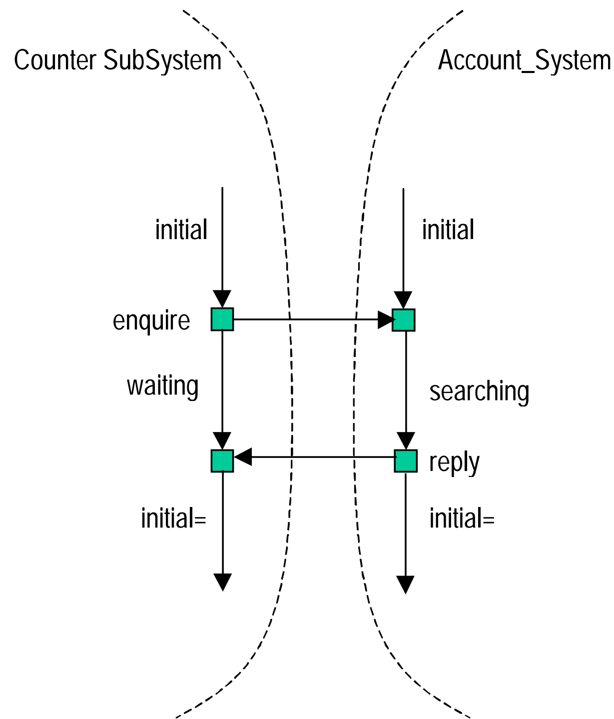

Figure 5: Banking Components

Figure 5 shows an extract from a RAD describing a part of the behaviour of our banking system. It shows how a Counter Subsystem in its initial state can initiate an enquiry of the Accounting Subsystem and that, following this action the Counter Subsystem waits for a reply from the Accounting Subsystem which eventually responds. The essential features of this piece of the behaviour are that both systems start in their initial states, the Counter Subsystem initiates the enquiry and waits for a response from the Accounts Subsystem, the Accounts Subsystem sends a reply, and both systems return to their initial states. In a final implementation of such a system, each of these interactions is likely to consist of an extended exchange of messages. However, to understand what is happening in the system there is no need for these details in the same way that the details of the exact nature of the connections is not needed in a diagram of the structure of the system. In fact too much detail at this stage of evaluation make the model obscure and confusing so it is preferable that only the essential features of the interactions is described.

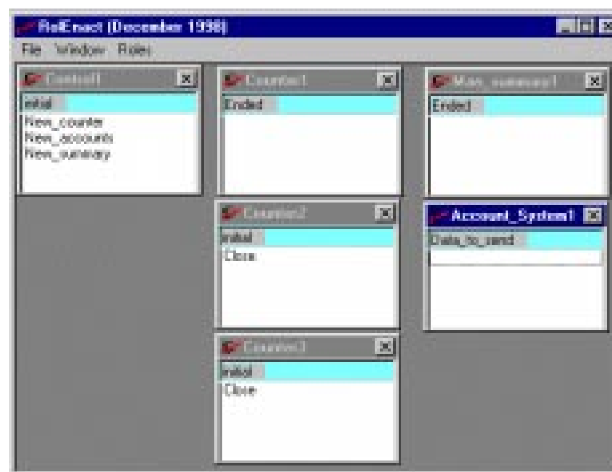

Figure 6: RolEnact Behaviour Analysis

In the first model which we made of the proposed system, a counter wishing to make an enquiry for a balance, or perform a transaction, interacted with the Accounts System and waited for the result. The Accounts Subsystem then sent revised data to the Summary Subsystem before returning to its initial state to wait for the next interaction. Examination of the completed model showed that a system constructed in this way works perfectly well when all of the components of the system are present. However, one of the requirements for the system was that it would continue to operate as Counter Subsystems opened and closed and whether the Management Subsystem was present not, but the behaviour of this system when the Management Subsystem is disconnected is not as required. There is no immediate change to the system when the Management Subsystem is closed and any Counter Subsystem is able to initiate an enquiry as before. The Counter Subsystem also receives the appropriate response. However, after giving its response to this enquiry, the Account Subsystem is unable to proceed as it is unable to pass the revised summary data to the Management System. The closure of the Management Subsystem leads to the entire system being halted but not until a Counter Subsystem has completed the next enquiry. What is happening is clear from the abstract model, but in a real implementation it could easily appear that it was the final enquiry that caused the failure. Further, in a real system, the Accounts Subsystem would have interactions with the data storage layer of the system and unexpected halting of the Accounts Subsystem could easily result in data corruption.

A slightly revised version of the system is shown as a RAD-like diagram generated by RolEnact in Figure 7. Here the interaction between the Accounts Subsystem and the Management Subsystem has been changed from one that occurs once for each interaction by the Account Subsystem with a Counter Subsystem to one that need not. Instead, it can occur at any time that the Accounts Subsystem is not busy processing an enquiry form a Counter Subsystem. This requires a revised implementation for the Accounts Subsystem so that it retains details of interactions not yet reported to the Management Subsystem it may also be appropriate for the detail of the interaction between the Account Subsystem and the Management Subsystem to be changed so that the whole of the accumulated data is passed to the Management subsystem. This revised system is not disabled when the Management Subsystem component is not present.

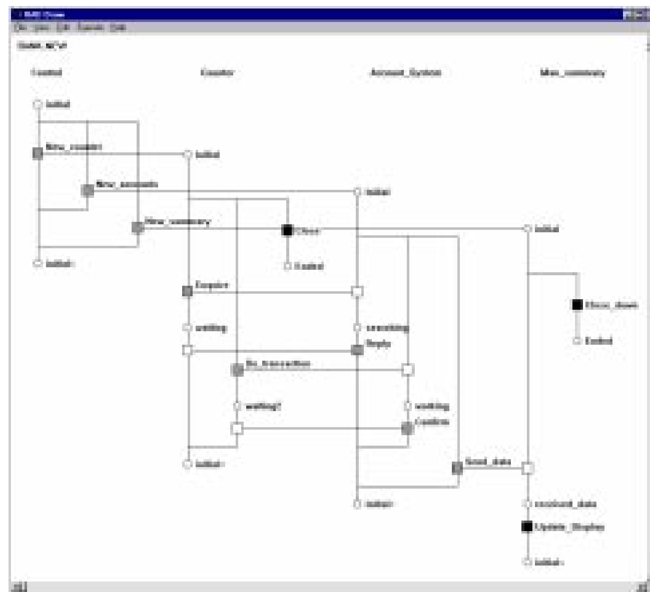

Figure 7: RolEnact generated RAD

Thus you can see how, by executing the model, and putting the design through a series of scenarios, we can validate it against both its own internal consistenct and against its ability to implement user requirements. This design validation will reduce the incidence of design flaws in the eventual implementation. The design validation scenarios can also be used to derive test cases for the eventual implementation in an obvious way.

RolEnact is nothing more than a way of representing communicating finite state machines, as we have already stated. But the notion of describing the Role of each component in a component-based

application, for the purposes of modelling and of validation, does seem a natural one. Components may be general in there full behaviour, but in a particular application they take on a particular role. The emphasis on specific behaviour encourages the designer to make an application oriented (business oriented) model of the proposed solution, which is exactly what is required for validation of customer requirements.

9. Conclusion

Although there is no doubt that the software systems we use today are larger, more complex, more highly featured and more reliable than systems in the past, the software industry has not been able to match the rate of progress in hardware. Formal methods offer the best solution where reliability is essential, but in their present state of development, they are not practical or economic unless the project is small or safety critical.

Building systems from components is a technique which is used extensively and effectively in hardware development and software component technologies are becoming available which offer the opportunity to build large systems rapidly by the effective reuse of code. Many systems lend themselves to development in this way as they have a number of generic types of behaviour, such as the management and storage of data which can be implemented using a database component. Like their hardware counterparts, many of the available components have been extensively tested and refined over many years of commercial use.

Systems built from components present a different verification problem from traditional systems. Typically, a new system is constructed from components which are themselves tried and tested, but are being used a novel way or in a combination or structure of components that is new. Consideration of the interfaces of components tells us whether it is possible to connect components in the manner proposed, but to know whether a proposed system makes sense, we need to examine and understand the behaviour of the components and their interactions. Formal methods provide a collection of techniques for this kind of analysis, but the size of the systems makes the application of these methods in the traditional way impractical.

A reasonable alternative which brings some of the advantages of formal methods to a component based project is to build and analyse a formal model which describes the behaviour of the system in an abstract way. Suitable abstractions will ensure that the analysis of the model is meaningful whilst removing unnecessary detail and keeping the size of the model within manageable bounds. A tool such as RolEnact can help with the construction of these models which can then be analysed in a manner similar to the way that hardware developers use their simulation tools to test and verify designs.

10. References

[1] G. Abeysinghe, P. Henderson, K.T. Phalp, and R.J. Walters, "An Audience Centred Approach to Modelling for Business Process ReEngineering," *5th International Conference on Re-Technologies for Information Systems (ReTIS 97)*, Klagenfurt, Austria, 1997.

[2] R.J. Allen and D. Garlan, "A Fomal Basis for Architectural Connection," *ACM Transactions on Software Engineering and Methodolgy*, 1997.

[3] M. Boman, J.A. Bubenko, P. Johannesson, and B. Wangler, *Conceptual Modelling*: Prentice Hall, 1997.

[4] D. Box and G. Booch, *Essential COM*: Addison Wesley, 1998.

[5] L. Cardelli, "Abstractons for Mobile Computation," Microsoft Research Technical Report MSR-TR-98-34, 1998, available at research.microsoft.com.

[6] A. Carpenter and N. Messer, "The use of VHDL+ in the Specification Level Modelling of an Embedded System," *International Forum on Design Languages*, Sepot, Switzerland 1998.

[7] S-K. Chin et al, "Formal Methods Tools to Support System Design," *First IEEE International Conference of Complex Systems*, 1995.

[8] E.M. Clarke, J.R. Burch, O. Grumberg, D.E. Long, and K.L. McMillan, "Automatic Verification of Sequential Circuit Designs," *Mechanical Reasoning and Hardware Design*, Royal Society Discussion Meeting, 1991.

[9] E.M. Clarke et al, "Model Checking and Abstraction," *ACM Transactions on Programming Languages and Systems*, 1994.

[10] M. Fowler, K. Scott, and G. Booch, *UML Distilled - Applying the standard Object Modelling language*: Addison Wesley, 1997.

[11] D. Garlan et al, “Architectural Mismatch, or, why it's hard to build systems out of existing parts,” *ICSE*, 1995.

[12] D.N. Gray et al, “Modern Languages and Microsoft's Component Object Model,” *Communications of the ACM*, vol. 41, 1998.

[13] O. Grumberg and D. Long, “Model Checking and Modular Verification,” *ACM Transactions on Programming Languages and Systems*, 1994.

[14] D. Harel, “On Visual Formalisms,” *Communications of the ACM*, 1988.

[15] D. Harel, “Statecharts: A Visual Formalism for Complex Systems,” *Science of Computer Programming*, 1987.

[16] M.M.K. Hashmi and A.C. Bruce, “Design and Use of a System-Level Specification and Verification Methodology,” *IEEE European Design Automation Conference*, 1995.

[17] M. Heimdahl and N. Leveson, “Completeness and Consistency in Hierarchical State-Based Requirements,” *IEEE Transactions on Software Engineering*, 1996.

[18] P. Henderson, “Laws for Dynamic Systems,” *International Conference on Software Re-Use (ICSR 98)*, Victoria, Canada, 1998.

[19] P. Henderson and R.J. Walters, “Component Based systems as an Aid to Design Validation,” *14th IEEE International Conference on Automated Software Engineering (ASE99)*, Cocoa Beach, Florida, 1999.

[20] P. Henderson and R.J. Walters, “System Design Validation Using Formal Methods,” *Tenth IEEE International Workshop on Rapid System Prototyping (RSP99)*, Clearwater, Florida, 1999.

[21] C.A.R. Hoare, *Communicating sequential processes*: Prentice-Hall International, 1985.

[22] C.A.R. Hoare, “How did Software get to be so reliable without proof,” *18th International Conference on Software Engineering (ICSE-18)*, 1996.

[23] S. Hodgson and M.M.K. Hashmi, “SuperVISE - System Specification and Design methodology,” *ICL Systems Journal*, vol. 12, 1997.

[24] G.J. Holtzmann, “The Model Checker SPIN,” *IEEE Transactions on Software Engineering*, vol. 23, 1997.

[25] C.N. Ip and D.L. Dill, “Verifying Systems with Replicated Components in Murphi,” *International Conference on Computer Aided Verification*, 1996.

[26] R. Kurki-Suoni, “Component and Interface Refinementin Closed-Systems Specifications,” *FM '99*, 1999.

[27] D.C. Luckman et al, “Specifiaction and Analysis of System Architecture using Rapide,” *IEEE Transactions on Software Engineering*, 1995.

[28] D.C. Luckman and J. Vera, “An Event-based Architecture Definition Language,” *IEEE Transactions on Software Engineering*, 1995.

[29] J. Magee and J. Kramer, *Concurrency: State models and Java Programs*: John Wiley and Sons, 1999.

[30] Object Management Group, “Common Object Request Broker: Architecture Specification,” .

[31] M.A. Ould, *Business Processes - Modelling and Analysis for Re-engineering and Improvement*: John Wiley and Sons, 1995.

[32] M.A. Ould, “Designing a re-engineering proof process architecture,” *Business Process management Journal*, vol. 3, 1997.

[33] K.T. Phalp, P. Henderson, G. Abeysinghe, and R.J. Walters, “RolEnact - Role Based Enactable Models of Business Processes,” *Information And Software Technology*, vol. 40, 1998.

[34] RolEnact, Available from: http://www.ecs.soton.ac.uk/~ph/RolEnact

[35] M. Shaw et al, “Abstractions for Software Architecture and Tools to Support Them,” *IEEE Transactions on Software Engineering*, 1995.

[36] M. Shaw and D. Garlan, *Software Architecture - Perspectives on an emerging discipline*: Prentice Hall, 1996.

[37] K. Sullivan and J.C. Knight, “Experience Assessing an Architectural Approach to Large Scale Reuse,” *18th International Conference on Software Engineering (ICSE-18)*, 1996.

[38] K. Sullivan, J. Socha, and M. Marchukov, “Using Formal Methods to Reason about Architectural Standards,” *19th International Conference on Software Engineering*, Boston, 1997.

[39] A. Thomas, “Enterprise JavaBeans Technology,” Patricia Seybold Group, White Paper prepared for Sun Microsystems Inc December 1998.