ELSEVIER

Information and Software Technology 39 (1997) 107–124

INFORMATION AND SOFTWARE TECHNOLOGY

Combining process modelling methods

Geetha Abeysinghe, Keith Phalp*

Department of Electronics and Computer Science, University of Southampton, Mountbatten Building, Highfield, Southampton SO17 1BJ, UK

Received 10 August 1995; revised 7 March 1996; accepted 25 April 1996

Abstract

This paper examines two modelling paradigms, namely Hoare's Communicating Sequential Processes (CSP) and a subset of Role Activity Diagrams (RADs) and shows how they can be combined to give a new approach to process modelling. We examine the two notations by reference to processes from two different business domains. For each domain, we transform a RAD model (by way of methodical mapping) to arrive at an equivalent formal CSP model. The latter is then explored using a stepper, which allows for process simulation by executing the model. The paper suggests that by providing a mapping between these notations we gain the accessibility of a well understood user-facing modelling paradigm, (RADs), whilst retaining the formality of CSP. This provides us not only with the advantages of understandable user-facing models, for process elicitation and presentation, but also gives us the ability to experiment with (by process simulation) the effects of process change.

Keywords: Business process modelling; Business process reengineering; Role activity diagrams; Communicating sequential processes

1. Introduction

Any business can be viewed as a collection of processes. These processes change as organizations evolve over time in response to their business environments. To keep ahead of the market competition, new ideas and change of business tactics have to be achieved quickly and efficiently. Process modelling has evolved as a technology for describing processes such that they may be understood and evolved with greater ease, and increased organizational visibility.

Within process modelling there are many methods and notations which may be used in order to describe the process under scrutiny. These methods range from formal (mathematical) rigorous notations, to more graphical (easier to understand) notations. Each of these kinds of notations has its own advantages and problems. Typically formal notations, may be executed on a computer and run (as programs) to study in detail the behaviour of processes. However, the main problem with such notations is that they are difficult to present to anyone other than an expert. Hence, it is difficult to validate process scenarios with users. In contrast, diagrammatic or graphical notations are excellent for process elicitation and presentation, since they may be understood with relative ease in a short space of time. However, they do not provide the benefits of rigorous process experimentation which can be gained with enactable notations.

In this paper we study two existing modelling paradigms, namely, a subset of Role Activity Diagrams (RADs), and Hoare's 'Communicating Sequential Processes' (CSP). These may be considered as best practice examples of both diagrammatic and formal notations. In studying these notations we make some comments about their use in modelling business processes, but the main motivation for our study is to arrive at a method of mapping from one paradigm to the other. The idea behind mapping from one notation to the next is to arrive at a coherent modelling method which will retain the advantages of both models, without having the associated weaknesses outlined above.

Our modelling method uses RADs, as a user-facing notation. This means that this notation is used in order to capture the process, and validate it with users. Having done this capture, the RADs are mapped to CSP for more rigorous experimentation with process, and with possible process changes. The results of this experimentation are fed back into new RADs which are again presented to users, and so on. Finally the new process is presented in RADs in order to educate process users about proposed changes. Hence, the formal notation

* Corresponding author. email:kp@ecs.soton.ac.uk, http://dsse.ecs.soton.ac.uk/~kp/

PII S0950-5849(96)01126-3

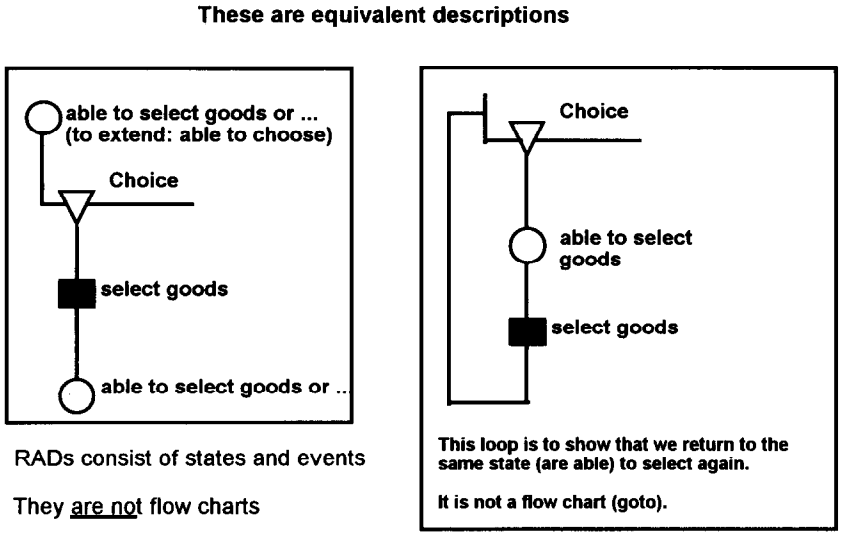


Fig. 1. Alternative descriptions of recursive behaviour

need not be presented to the users, but can be used to test the logic of the process with some rigour.

We present two kinds of process examples to illustrate our methods. We start with a simple example, 'shopping', which is a simplified view of the retail process. We use this example as a common reference point in order to describe the elements of both notations, and to introduce our rules for mapping from RADs to CSP. We then present a more complex example, which describes the upstream activities of an industrial software development process. We use the latter (real example) in order to test our mapping, and to show that it remains valid for a genuine industrial process.

2. Background to the modelling paradigms

Role Activity Diagrams are a notation originally developed for software process modelling [1]. In the UK they have been used and promoted by both Praxis [2] and Co-Ordination Systems [3], and their merits have been discussed at a number of tutorials and meetings on process modelling — notably those supported by the IOPTClub [4]. A CASE tool for process modelling RADitor [5] marketed by Co-Ordination systems uses Role Activity Diagrams as its diagramming method. RADs can be considered to be a state of the art single paradigm process modelling approach, and are well known among the process modelling community (particularly in the UK).

CSP is a programming language based on concurrency and communication introduced by Hoare in 1978 [6]. In this paper we view CSP as a process modelling paradigm rather than as a programming language. CSP has two forms, event CSP and channel CSP. We will be considering only event CSP (and will be referred to as CSP from here on) in this paper. Hoare's CSP has been implemented in a stepper [7] using the executable specification language Enact [8]. The existence of the stepper is an added advantage in using CSP. We have simulated the parallel behaviour among processes using the CSP stepper.

2.1. An example process: shopping

We choose as our first example process 'shopping'; how retailers process their customers. This is to allow our initial choice of application domain to be something familiar to all our readers. In our scenario a customer having entered the shop should be able to select goods and then pay for what s/he has selected, return goods and then get a refund for the goods returned, or leave the shop. During the time the customer is in the shop, s/he should be able to select and/or return goods any number of times s/he desires to until the customer decides to leave the shop. The shop should be able to display goods for the customer to select and manage the transaction of receiving money for those goods.

2.2. Elements of the modelling notations

In describing the elements of the modelling notations we will make reference to our 'shopping' example and present descriptions of shopping in both RADs and CSP.

2.2.1. Roles and activities versus processes and events

Roles and activities. The central concept of Role Activity Diagrams is that of a role. A role describes a sequence of steps or activities which can be acted out by a person or perhaps by a group or department. Roles are acted out in parallel and communicate through interactions (see below). It is important to realize that a

role is merely a type. For example, it may describe the behaviour of a class of people. So, a role may describe the responsibilities and interactions of a manager, or a cashier or some other role[1]. A single instance of a role can be acted by many people, and similarly a single person may act many role instances. For example, one person may act as a project manager role and also as an engineer role.

A role has a thread of activities (represented by square boxes) within it, and is read from top to bottom, activities being connected by state-lines (the state between them). The intention is for the notation to be much more akin to Finite State Machine [9,10] or to Petri-net [11] approaches than it is to flow charts, and some authors use a circle to label states in order to further emphasize this distinction [2,5,12].

Indeed, this use of a circle to label states is a convention which we will adopt in this paper. Furthermore, we will always avoid using loops, preferring to use these state labels to show the way roles can return to previous states (see Fig. 1).

There are two kinds of activities within a role, actions and interactions. In Role Activity diagrams an action is a process step that the actor of the role carries out in isolation. Thus actions do not involve any joint behaviour with another role. An action changes the state of the role in which it occurs. Actions are represented by a shaded (we have shown as black) square. An interaction between two roles implies that they have some shared or joint behaviour, and is represented by joining activities (left unshaded) within different roles by a horizontal line. An interaction may change the state of any of the roles which are involved in that interaction.

Processes and events. The main concept of CSP is a process. A process describes how an object behaves. The set of events which a process participates in, is known as its alphabet; the alphabet is a process P is denoted as αP (in our notation). A process is defined in terms of the events in its alphabet by defining the allowable sequences of events. In CSP an event is assumed to be instantaneous or, in other words, it is an action which does not take any time to occur.

In describing CSP we adopt the following convention: event names start with lowercase letters, process names, and variables denoting processes start with uppercase letters. For example, in 'shopping' an example of a process is a customer which may have the following alphabet:

$$\alpha\texttt{Customer} := \{\texttt{enter}, \texttt{select}, \texttt{payment}, \texttt{return}, \texttt{leave}\}.$$

[1] Note that a role may not always be a person. It may, for example, describe the behaviour of a system which interacts with people.

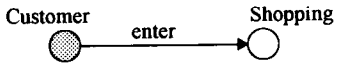


Fig. 2. 'Customer' process = enter followed by 'Shopping' process.

A process is described by the sequence, (event → Process). The operator '→' denoting sequence. The customer process in 'shopping' can be described (simplified) as,

$$\texttt{Customer} := \texttt{enter} \rightarrow \texttt{Shopping}$$

The above describes a process Customer which first executes (or participates in) the event of entering the shop, enter, then executes the process of Shopping. The process Shopping may constitute of a number of events such as, selecting goods, paying for the goods, and so on.

We can show this pictorially as in Fig. 2.

In the pictorial view we represent the processes by circles (the process being defined, for example, process Customer in the above picture, as a filled circle. Other processes for example, Shopping in Fig. 2, as unfilled circles) and events by the named connecting arrows.

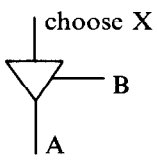
It should be noted that the operator '→' always takes an event on its left and a process on its right. The sequential combination of processes on the other hand is described by the operator ';'. For example, if P and Q are two processes, the combination (P;Q) describes a process which first behaves as P, when P terminates successfully, then behaves as Q. If P does not terminate successfully the behaviour of Q is never executed. The successful termination of a process is represented by 'SKIP' (this will be further described in Section 2.2.3). For example, in the 'shopping example we can describe the process Shopping by two sequential processes as:

$$\texttt{Shopping} := \texttt{Select\_Goods}; \texttt{Leave}$$

where Select_Goods is the process of selecting and paying for goods and Leave is the process of a customer leaving the shop. According to the above description the customer has to successfully complete the process of selecting before he is able to leave.

2.2.2. *Alternate or choice*

Role Activity Diagrams have two constructs for showing alternative or parallel paths within a role. Alternative paths are where the choice is dependent on some (yes-no) condition. This construct is usually denoted by an inverted triangle. The following denotes:



if X is chosen then, follow path A, else follow path B.

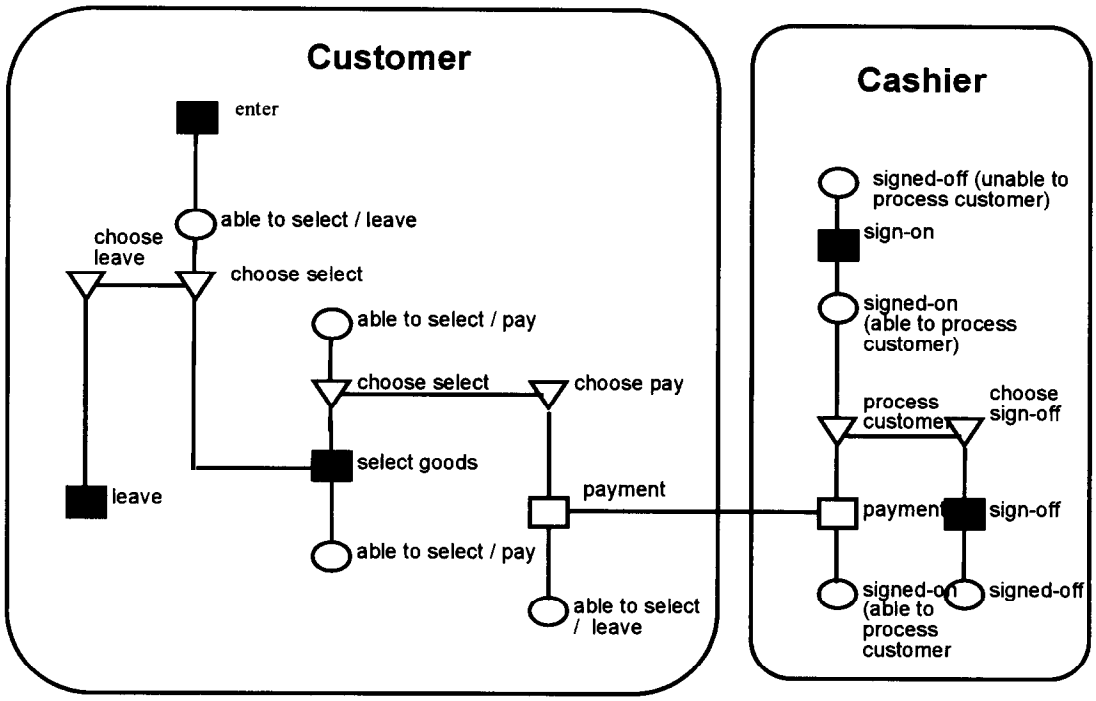


Fig. 3. Role activity diagram of a 'Shopping' process. The diagram shows two roles: customer and cashier for a retail outlet, e.g. a supermarket. Having entered the customer may choose to select goods or leave. Once goods have been selected the customer must make a payment before leaving. However, a number of selections can be made before paying. On payment there is an interaction with the cashier. This is only possible if there is an instance of a signed-on cashier for the customer to interact with.

A simple example of a customer interacting with a cashier is represented in Fig. 3.

After the customer enters the shop s/he is faced with the choice of leaving the shop or selecting goods. This is represented by the two inverted triangles named 'choose leave' and 'choose select/return' respectively. Once the customer has done a selection (action 'select') s/he has the choice of selecting more goods (the alternate 'choose select') or paying for what is already selected (alternate 'choose pay'). Ould refers to such alternative courses of action in a RAD as '*case refinement*', refining the state of the process according to different cases [2].

In CSP a simple choice is described by the operator '|'. This allows the user to define alternate behaviours of an object. For example, taking the simple customer example shown in Fig. 3, a customer after entering the shop can either do shopping (that is select goods and pay for them) or leave. This scenario can be described by a simple choice as given below.

```
Customer := enter → Shopping

Shopping     :=  Select_Pay
               |  Leave
```

The effect of the choice operator '|' is that when one path is chosen the process is committed to pursue that path; in other words all other paths in that choice become inaccessible.

The CSP description above can also be represented by a state-event diagram (see Fig. 4).

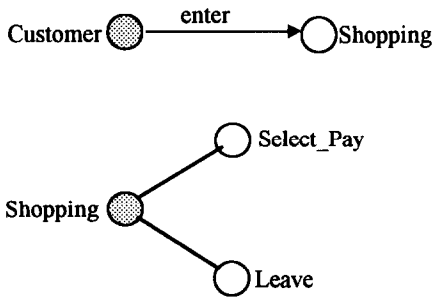


Fig. 4. 'Shopping' process = leave or choose to select and pay.

2.2.3. Parallelism

RADs display two kinds of parallelism; the role instances acting in parallel, and the threads of parallel activities within a role. In the 'shopping' example in Fig. 3, we can identify two parallel roles, cashier and customer. Role Activity Diagrams assume no ordering on the way instances of roles proceed. In other words, the RAD describes the behaviour of the role, and its relations to other roles, but it does not describe the allocation of resources to roles, or the number of roles active at any one time, and so on. For example, an instance of a cashier role may be acted out by the same person who previously acted as a supervisor, but this may happen in parallel with another instance of the

supervisor role. Similarly there will be a number of cashier roles acting in parallel at any one time. Roles are not descriptions of instances of behaviour rather they describe a type of behaviour to be acted by the role. With Role Activity Diagrams the parallel construct is used to show the behaviour within a role, not to co-ordinate them (as CSP does with processes).

We can also have parallel (or concurrent) threads of activities within a role. These parallel vertical threads are denoted by the ordinary triangle symbol, △. Ould calls this *'part refinement'*, refining (or dividing) the state of the role into a number of separate parts [2]. There is no choice here, and thus no forcing down one thread. Indeed, it is assumed that all paths are taken. We will further examine parallel threads in our second (more complex) shopping example.

CSP gets its strength from its ability to describe processes that can be executed in parallel. The convention used to represent parallel composition in CSP is '||'. A complex process can be described as a number of simple processes running in parallel. When two or more processes are executed in parallel the processes synchronize in their shared events. That is, where two processes have the same event in their alphabets they both must execute that event simultaneously [13]. Therefore, CSP is said to support broadcast communication.

If we take a more complex example of 'shopping' where the customer is able to return goods then the customer can be described elegantly by,

$$\texttt{CUST} := (\texttt{enter} \rightarrow \texttt{Shopping}) \ ; (\texttt{exit} \rightarrow \texttt{CUST})$$

The process Shopping can be described as,

$$\texttt{Shopping} := (\texttt{SelectGoods})^* \ || \ (\texttt{ReturnGoods})^*$$

In the above description '*' means, 'may be executed zero or more times'. Therefore, the above CSP description implies that a customer after entering a shop is able to 'select goods and then select more or pay' and/or 'return goods and then get refund' zero or more times before deciding to leave the shop. In this case we define the processes SelectGoods and ReturnGoods as given below:

$$\begin{aligned} \texttt{SelectGoods}_0 &:= \texttt{select} \rightarrow \texttt{SelectGoods}_1 \\ \texttt{SelectGoods}_1 &:= (\texttt{select} \rightarrow \texttt{SelectGoods}_1) \\ &\quad | \ (\texttt{payment} \rightarrow \texttt{SKIP}) \end{aligned}$$

This (above) description implies that once the customer has done a select event s/he is faced with the choice of selecting more or paying for the selected goods. The process of returning goods can be described as:

$$\texttt{ReturnGoods} := \texttt{return} \rightarrow \texttt{payback} \rightarrow \texttt{SKIP}$$

This (above) description describes the behaviour of the customer who returns one or more items and gets a refund. This sequence of events can be repeated any number of times until the customer decides to leave the shop. Once the customer leaves the shop s/he is able to behave in a similar manner once again, indicated by '(exit → CUST)'.

In the simple 'shopping' example introduced in Fig. 3, we can describe two processes, the customer and the cashier. In Section 2.2.2 we described the Customer as:

$$\begin{aligned} \texttt{Customer} &:= \texttt{enter} \rightarrow \texttt{Shopping} \\ \texttt{Shopping} &:= \texttt{Select\_Pay} \\ &\quad | \ \texttt{Leave} \end{aligned}$$

The customer may enter the shop, select goods and pay for them or leave the shop. This behaviour is described by the processes Shopping and Leave respectively.

$$\begin{aligned} \texttt{Select\_Pay} &:= \texttt{select} \rightarrow \texttt{Select\_Pay} \\ &\quad | \ \texttt{payment} \rightarrow \texttt{SKIP} \\ \texttt{Leave} &:= \texttt{leave} \rightarrow \texttt{SKIP} \end{aligned}$$

The cashier may sign on, receive payment from the customer or sign off. These two simple processes can be described in CSP as follows:

$$\begin{aligned} \texttt{Cashier} &:= \texttt{signOn} \rightarrow \texttt{Signed\_On} \\ \texttt{Signed\_On} &:= (\texttt{payment} \rightarrow \texttt{Signed\_On}) \\ &\quad | \ (\texttt{signOff} \rightarrow \texttt{Cashier}) \end{aligned}$$

In the shop the two processes, Customer and Cashier, will be carried out concurrently. In CSP we define this as:

$$\texttt{Shop} := \texttt{Customer} \ || \ \texttt{Cashier}$$

When executing the process Shop, Customer and Cashier should synchronize in the shared event 'payment'. In other words the Customer cannot pay for the goods until the Cashier is ready to participate in the event payment, that is, the cashier should be in the state signed-on (Fig. 3). This synchronization is also indicated in the RAD, the difference being that RADs synchronize between roles.

The parallel mechanism in CSP in conjunction with communication on shared events can be used to control complex communicating processes. However, the parallel threads within a RAD are more akin to the use of the parallel operator within a process than its use to control separate processes.

2.2.4. Interactions and shared events

We have noted that an interaction in Role Activity Diagrams may change the state of any role which participates in that interaction. For example, in Fig. 3, the interaction 'payment' changes the state of both roles, customer and cashier. Before the payment the customer may either select more goods or pay for goods, but may not leave. After payment the customer may select again or leave. The payment activity changes the state

of the cashier so that s/he can process the next customer. The shared (interaction) of activities must take place synchronously. This synchronization may take place over time, and may be quite complex. We are allowed to represent complex interactions, for example, gaining agreement, with this same construct. Another RAD would be used to examine the details of such an interaction.

The equivalent of an interaction in CSP is where parallel processes co-ordinate on shared events. For example, the exchange of money for goods between a cashier and the customer (which is a synchronous interaction) can be described (over-simplified for clarity) as:

```
Cashier  := signOn  → (payment → ...)
Customer := enter   → (select  → payment → ...)
```

When the two above processes are executed in parallel the shared event, 'payment' must occur at the same time thus co-ordinating the two processes. For example, according to the above description a customer can enter and select at any time but will be able to pay only after a cashier signs on. Similarly a cashier, after signing on, can accept payment only after the customer has finished selecting.

3. Mapping between CSP and RADs
Part one: shopping

Here we examine whether we can map between equivalent constructs in both notations, specifically whether we can take a RAD and describe it in CSP. We first illustrate our mapping ideas with reference to versions of the shopping example. We introduce a subset of our mapping rules and show the original RAD descriptions and the equivalent CSP. We will then go on to describe how our mapping works for our example of a portion of the software development process, which describes the upstream (requirement activities) of an industrial software developer.

We have used a CSP process to be equivalent to a RAD role. If we take the shopping example, the customer process can be decomposed into events, just as the role can be decomposed into actions. However, CSP supports high levels of abstraction, in that a process can be defined in terms of other processes. For example, the process Shop can be defined as the parallel composition of the two processes, Customer and Cashier, given in CSP as,

```
Shop = Customer || Cashier
```

A process which exhibits the same behavioural pattern repeatedly is defined in CSP using recursion. Taking the Customer process in our 'shopping' example; after leaving the shop the customer can display the same behavioural pattern again, that is enter a shop and do shopping. We can describe such behaviour as:

```
Customer := (enter → Shopping); Customer
```

Similarly, in Section 2.2.3 we represented the fact that once the customer has selected goods, s/he can select more or pay for the selected goods by,

```
Select_Pay := select → Select_Pay
            |  payment → SKIP
```

In RADS this repeated behaviour is represented by using a state label for the point of return. In a RAD a state can represent potential future behaviour, or an indication of the past behaviour, or both. The above CSP is represented in Fig. 3 by repeated returns to the state 'able to select/pay', after each occurrence of selecting goods (the action 'select goods').

A further distinction between the notations is that an interaction in a RAD can occur over time, e.g. we might have an action 'reaching agreement' whereas CSP events are usually thought to be instantaneous. If we wish to represent a non-instantaneous event in CSP, we normally define two separate events; one denoting the start of the activity, the other denoting the completion [13]. Despite these arguments we can construct an equivalent description in both paradigms. We first use a simple example of a customer interacting with a cashier, as introduced in Fig. 3.

In order to make our mapping explicit we will now describe the initial rules which we use to map from a RAD to an equivalent description in CSP. (Note that we introduce further rules later in our software development example. However, in the interests of clarity we will only introduce rules as required.) We will then apply these rules to our simple RAD process model in order to arrive at our CSP model.

(1) Roles become higher level CSP processes running in parallel.
Therefore we get,

```
Shop = Customer || Cashier
```

(2) We read each role by stepping through states from top to bottom, converting each action to a CSP event. We thus get the following alphabets:

αCustomer:= {enter, select-goods, payment, leave}

αCashier:= {sign-on, payment, sign-off}

(3) Each event (action) moves the process (role) to its next state. At each point where we encounter a state which represents a point of return we create a new sub-process with a name representing that state. In the case where the state is the first construct of the role, then the process takes the name

of the role itself. Thus, we can describe the process Cashier as,

```
Cashier  := sign-on → Signed_on
Customer := enter → Select_Leave
```

(4) The RAD alternate construct (an inverted triangle) becomes a simple choice in CSP. Each alternative is converted into an individual process. The name of the process is taken from the given choice. Hence the first choice the customer has after entering the shop is to 'choose to select' or 'choose to leave', and these are represented by the sub-processes (of Customer) 'Leave' and 'SelectGoods' respectively. The Select_Leave process is therefore written as:

```
Select_Leave  :=  Leave
                | SelectGoods
```

where,

```
Leave       := leave → Customer
SelectGoods := select-goods → SelectGoods
             | Pay
Pay         := payment → Select_Leave
```

Similarly, the cashier, having signed-on, may choose to 'process customer' or 'sign-off', and these can again be represented by separate processes, namely 'ProcessCustomer' and 'SignOff'. The process Signed_on can be then described as,

```
Signed_on       := ProcessCustomer
                 | SignOff
SignOff         := sign-off → Cashier
ProcessCustomer := payment → Signed_on
```

(5) Each parallel path also becomes an individual CSP process (sub-process). However, these processes will run in parallel, rather than being alternatives. Parallel paths enable the role to be in any of the states implied by each parallel thread.

If the leaf state of a parallel thread denotes the returning to a point outside the thread, then that state is converted to a successful termination of that thread (the process SKIP in CSP).

For this example we have no parallel paths in the roles.

(6) Interactions become shared events. The common event gets included in the alphabets of all processes involved in the interaction.

The two high-level processes, Customer and Cashier, share the event 'payment'. When the two processes are executed in parallel, they synchronize on this shared event.

Before the 'payment' the customer is in the state 'able to select/pay'. In order for 'payment' to occur the cashier must also be ready to participate in the 'payment' event, that is the cashier must be 'signed-on'. When the event 'payment' happens the customer moves to the state 'able to select/leave' (note that this initial state is represented in our CSP by the process 'Customer'). The cashier also moves to the next state, in this case, 'signed-on (able to process customer)'.

Putting these rules together we get the following CSP.

```
Customer     :=  enter → Select_Leave

Select_Leave :=  Leave
                | SelectGoods

Leave        := leave → Customer

SelectGoods  := select-goods → SelectGoods
                             | Pay

Pay          := payment      → Select_Leave
```

```
Cashier        :=  signon →    Signed_on

Signed_on      :=  ProcessCustomer
                 |       SignOff

ProcessCustomer :=  payment →  Signed_on

SignOff        :=  signoff →   Cashier
```

Using the CSP stepper we can execute the CSP model given above. For example, we can execute in parallel the Cashier and the Customer processes, given in CSP as:

```
Shop := Cashier || Customer
```

The logical of the business process model represented by a RAD can thus be tested by mapping it into a CSP description. Similarly, a CSP description which is slightly difficult to understand by the layman can be represented by a more easily understandable RAD. The two paradigms complement each other in this way.

We now consider a more complex shopping example to further test our method for mapping from RADs to CSP. Applying the rules of mapping we get as before:

```
Shop := Customer || Cashier
```

where each process will have the following alphabets (Rule 2)

```
αCustomer := {enter, leave, select, payment,
              payback} and
αCashier  := {sign-on, payment, payback, sign-off}.
```

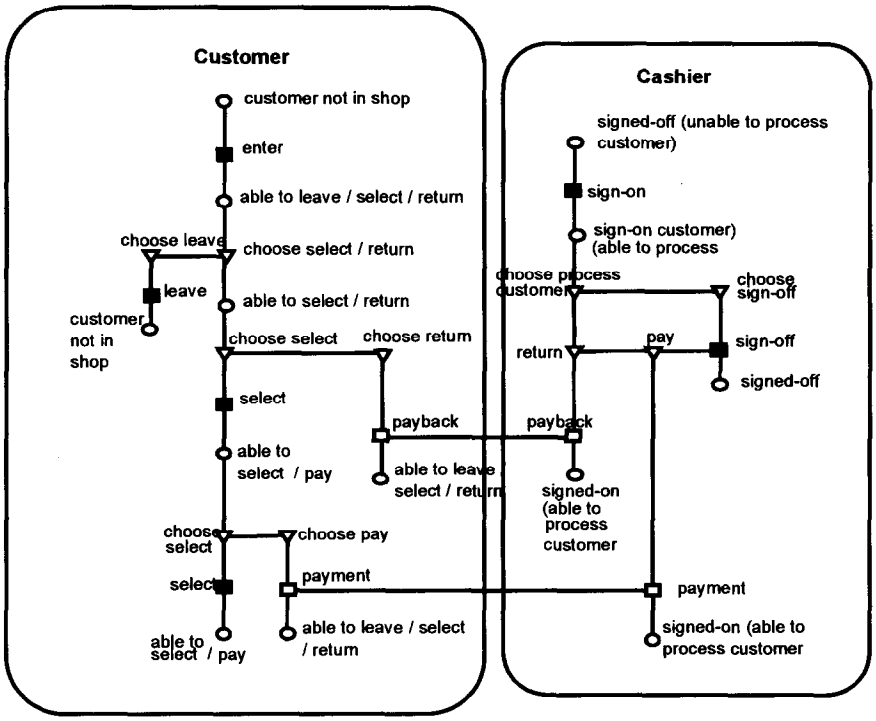


Fig. 5. More detailed role activity diagram for 'Shopping'.

We will now consider the remaining rules taking each role in turn.

Customer Role

Applying Rule 3, we can describe the process Customer in CSP as:

```
Customer := enter → Leave_Select_Return
```

Dealing with alternate threads, as given in Rule 4, we describe the process

```
Leave_Select_Return as,

Leave_Select_Return :=   Leave
                       | Select_or_Return
```

where the process Leave as described for the previous example is,

```
Leave := leave → Customer.
```

Dealing with alternate threads (as given in Rule 4) we describe the process,

```
Select_or_Return as:

Select_or_Return := (Select | ReturnGoods)

Select := select → Select_or_Pay          (Rule 3)

Select_or_Pay := SelectAgain              (Rule 4)
                | Pay

SelectAgain := select → Select_or_Pay

Pay := payment → Leave_Select_Return
```

The process Return is described as,

```
ReturnGoods := payback → Leave_Select_Return
```

```
Shop            := Customer || Cashier

Customer        := enter → Leave_Select_Return

Leave_Select_Return :=   Leave
                       | Select_or_Return

Leave           := leave → Customer.

Select_or_Return := Select | ReturnGoods

Select          := select → Select_or_Pay

Select_or_Pay := SelectAgain
                | Pay

SelectAgain :=  select → Select_or_Pay

Pay         :=  payment → Leave_Select_Return

ReturnGoods :=  payback → Leave_Select_Return
```

Cashier role

The Cashier is described as,

```
Cashier := sign-on → Signed_on.
    Signed_on      :=    Process_Customer
                   |     Sign_off
    Process_Customer     := Pay | Return
Pay       := payment →     Signed_on
Return    := return →      Signed_on
Sign_off  := sign-off →    Cashier
```

```
Cashier          :=   sign-on → Signed_on

Signed_on        :=   Process_Customer
                      |      Sign_off

Process_Customer      :=     Pay
                             |  Return

Pay       :=   payment →     Signed_on

Return    :=   payback →     Signed_on

Sign_off  :=   sign-off →    Cashier
```

4. Mapping between CSP and RADs Part two: launch process

4.1. The mapping experiment process

In order to test our mapping more thoroughly we have chosen to model a more complex process. Rather than choose an artificial example, we have chosen to model part of the upstream (requirements-based) activities of a real software development process.

An immediate problem with using RADs for such a process is that the diagrams soon become large and unwieldy. Hence, we have chosen to split the process into four linked RADs. Two of the RADs (Figs. 6 and 8) describe the higher level elements of the process. These are linked by RADs (Figs. 7 and 9) which show the details of two important interactions. In addition, Fig. 7 shows the link between Figs. 6 and 8, i.e. 'project team set up' and 'approved project' respectively.

In using this idea we have tried to be consistent with the way interactions in RADs work. The interaction here must still be synchronous, and will result in a change of state of all of the roles involved in the interaction. However, the choice of which next states the roles move to is dependent on the interaction.

By using such an approach we overcome two problems

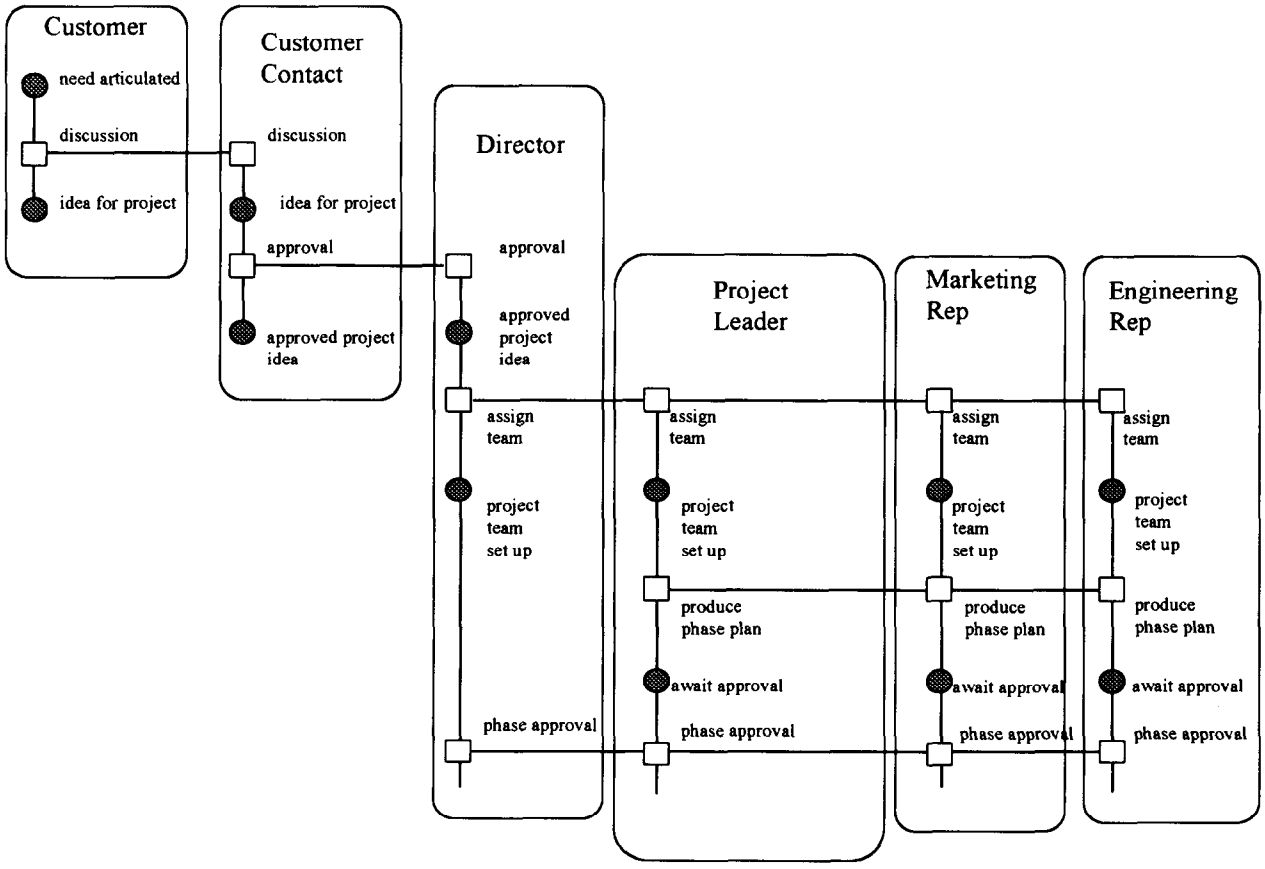


Fig. 6. Concept phase of project launch.

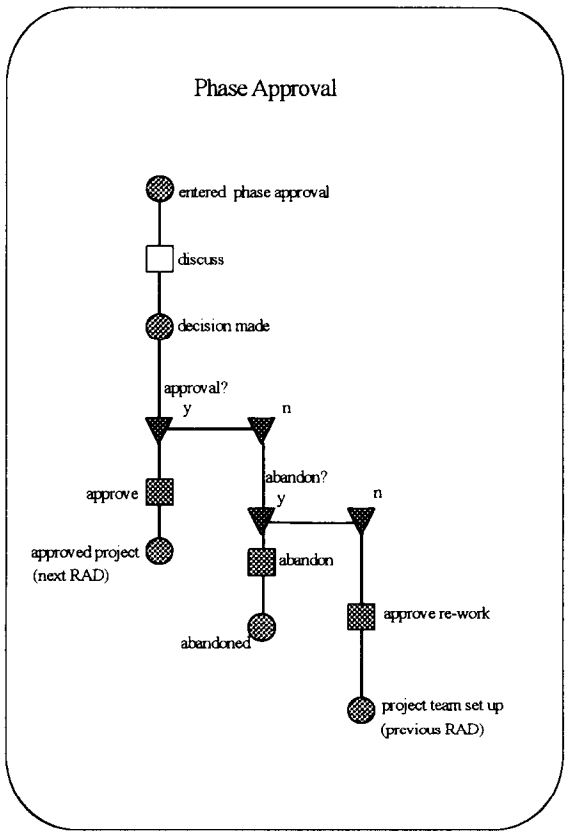


Fig. 7. Phase approval.

traditionally associated with RADs. Firstly, we are able to split a complex diagram into manageable chunks. We have found that in using RADs to analyse business processes, the size and scale of processes has resulted in very large and complex diagrams. Though RADs are not intended to map the process at all levels of detail in a single whole, it is still very easy to produce complex depictions of processes, particularly since low level details may sometimes have an impact at higher levels of abstraction.

Secondly, we are able to use a hierarchical structure to examine some of the important detail of certain key interactions (see our rule number 3 in Section 3), and to structure together individual RADs into a coherent whole. As an added bonus, we can illustrate our mapping in more accessible or manageable chunks. Hence, we will start by considering each figure (each separate RAD) in turn.

In order to further experiment with mapping between paradigms we have attempted to increase the consistency of our RADs by formulating some guidelines for drawing our diagrams.

4.2. Guidelines for drawing RADs

(1) Any activity or interaction causes a change of state. Therefore, we follow each activity by a corresponding state label. However, the activity and state label may be separated by a decision point (i.e. a choice).
(2) Consequently, any change of state must be preceded by an activity (or by a decision point). Again the activity and state label may be separated by a decision point.
(3) Complex interactions which may result in multiple changes of state[2] (e.g. review meetings) may be represented by a separate RAD. In order to denote this, the activity will have thread lines coming from it to show that the role continues elsewhere. An example of this is given for the interaction (phase approval) in Fig. 6. This new RAD may also be the link to another RAD. e.g. Fig. 8 (phase approval) links to Fig. 9 through the state 'approved project' or return to Fig. 7 through the state 'project-team-setup'. This stream of choices will not change our mapping rules. That is, we will still form new processes for each choice, and for each subsequent state which is a point of return.
(4) We limit ourselves to a subset of RADs, using roles, states, actions, interactions, parallel threads and chosen threads.
(5) We interpret choice to be yes-no or to be multi-choice, e.g. choosing one of three threads.
(6) We interpret parallel to mean concurrency over some time period. That is that all paths will be followed. If all paths do not have to be followed this implies that a choice may be made, hence, choice can be used.

4.3. Mapping rules reminder and extensions to rules

(1) Roles become higher level CSP processes running in parallel.
(2) We read each role by stepping through states from top to bottom, converting each action to a CSP event.
(3) Each event (action) moves the process (role) to its next state. At each point where we encounter a state which represents a point of return we create a new sub-process with a name representing that state. In the case where the state is the first construct of the role, then the process takes the name of the role itself.
(4) The RAD alternate construct (an inverted triangle) becomes a simple choice in CSP. Each alternative is converted into an individual process. The name of the process is taken from the given choice.
(5) Each parallel path also becomes an individual CSP

[2] A simple change of state is where the interaction moves all roles involved in the interaction from their previous state to the next single state (i.e. a before and after). A multiple change of state is where the interaction may move (all) the roles to one or more new states (but the same new state for all roles) depending on the details of the interaction — as shown in a separate RAD.

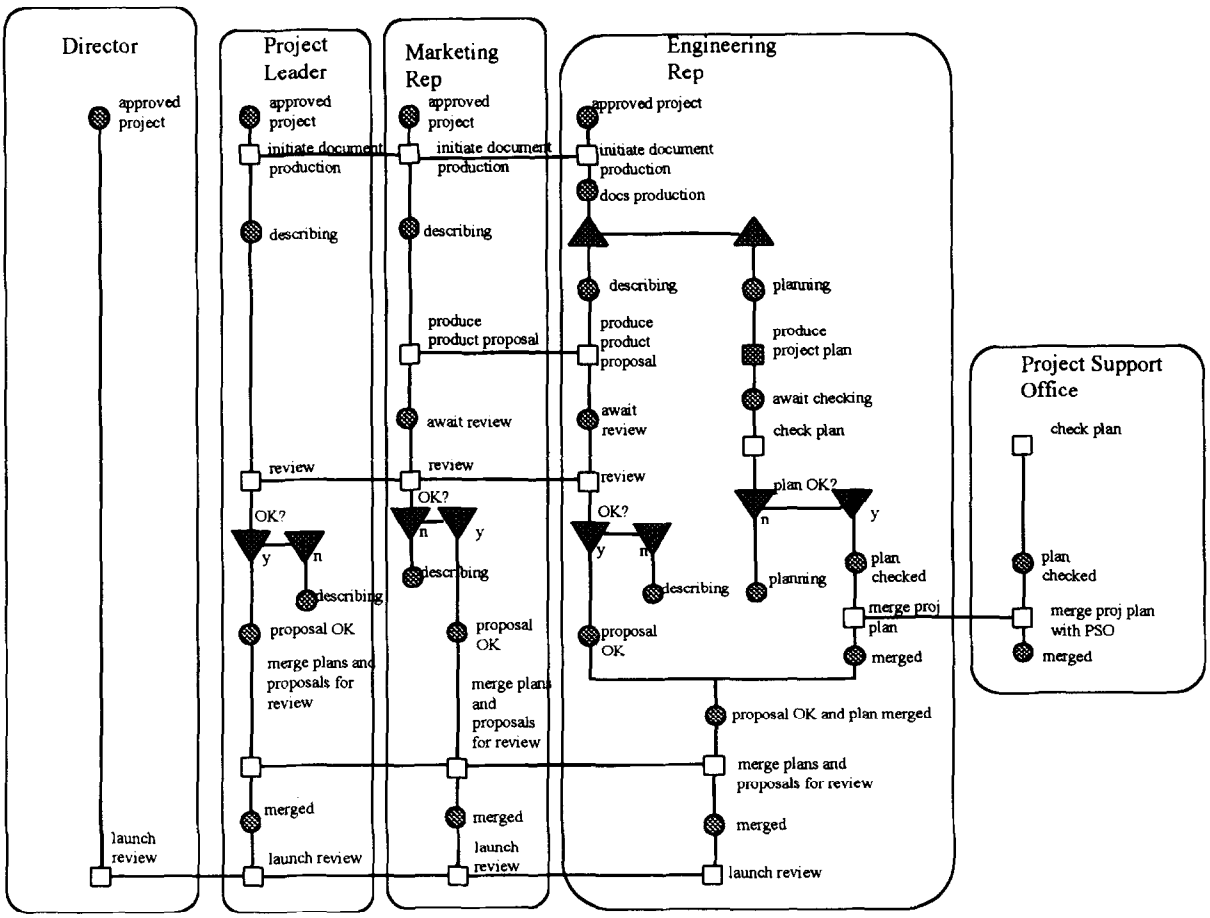


Fig. 8. Feasibility phase of project launch.

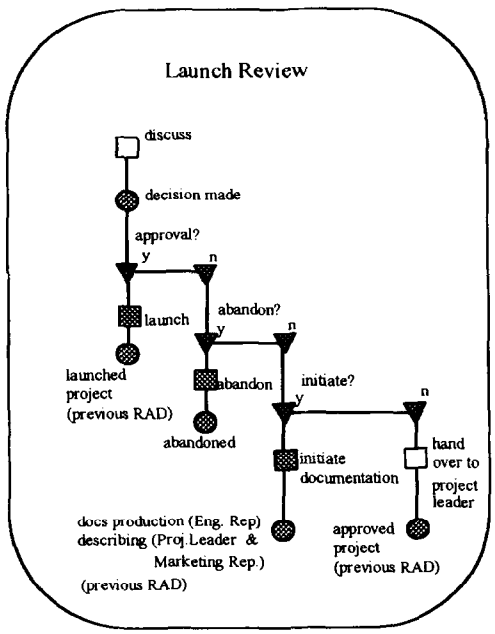


Fig. 9. Launch review.

process (sub-process). However, these processes will run in parallel, rather than being alternatives. Parallel paths enable the role to be in any of the states implied by each parallel thread. If the leaf state of a parallel thread denotes the returning to a point outside the thread, then that state is converted to a sequence of two processes; the successful termination of the parallel thread (SKIP in CSP), followed by the process representing the point of return.

(6) Interactions become shared events. The common event gets included in the alphabet of both processes (roles). However, for complex interactions which are expanded in another RAD we use Rule 9 below.

New rules

(7) A process P is described as:

$$P := e \rightarrow Q$$

where e is an event and Q is a process.

If a role R has only one action e, then, that role is converted to a process P, where P is described as:

$$P := e \rightarrow SKIP$$

The process SKIP in CSP denotes the successful termination of a given process.

(8) When we map a RAD into CSP, we create a new process for each point of return. This means that in some cases we may arrive at a CSP description of the format:

A := B

Where A is the role name and B is the point of return we do not create a process with the role name (since, this would cause duplication) rather we create a process with the name of the point of return. Similarly, if A is a point of return and B is an interaction we ignore B, and create a process with the name A — the point of return.

(9) Complex interactions are expanded to form another RAD. For the purposes of our CSP, we regard the interaction as a sub-process.

(10) Shared states which are also points of return will be converted into sub-processes. However, where these sub-processes are not actually the same we will denote their distinct identity by using the name of the role as a prefix.

(11) In CSP, events and processes should have unique names. In order to avoid having clashing names for the processes which denote roles in our different diagrams, we adopt a naming convention which adds a prefix (based on the name of the new RAD) when we encounter a role name which would otherwise become overloaded. The exception to this rule is where we expand an interaction to give us another role. This is because the roles involved in the interaction are already described within the original RAD.

(12) When our mapping produces two CSP processes which follow each other without an intervening event (e.g. A → B), the arrow is converted to ";" which denotes sequence in CSP. That is, we specify that the first process must end before the second commence.

4.4. The mappings for each Role Activity Diagram

We now attempt to apply these rules to our RADs which represent the upstream activities of a software development process. For simplicity, we have described the RAD using four different figures. Fig. 6 and Fig. 8 show the main RAD, whereas, Figs. 7 and 9 represent two expanded interactions. Taking each figure in turn and applying the rules, we will now attempt to arrive at the CSP descriptions.

As we go through our rules we give examples of where they are applied.

(1) Rule 1 — Roles become processes running in parallel.

```
Mapping := Customer||Customer_Contact||
           Director||
           Project_Leader||Marketing_Rep||
           Engineering_Rep
```

(2) Rule 2 — The actions each role participates in constructs the alphabet of the corresponding process.

αCustomer :=	{discussion}
αCustomer_Contact :=	{discussion, approval}
αDirector :=	{approval, assign-team}
αProject_Leader :=	{assign-team, produce-phase-plan}
αMarketing_Rep :=	{assign-team, produce-phase-plan}
αEngineering_Rep :=	{assign-team, produce-phase-plan}

(3) Now we will consider each role in turn. Taking the role 'Customer' and applying Rule 3 we get:

```
Customer := discussion → SKIP
```

Similarly for Customer_Contact we get:

```
Customer_Contact := discussion
  → (approval → SKIP)
```

Taking the Role Director we arrive at the following:
Applying Rule 4: states which are return points become sub-processes. Hence:

```
Director := approval
  → (assign-team → Project_Team_Setup)
```

Using Rule 9

```
Project_Team_Setup := Phase_Approval
```

Note that phase approval is then expanded in Fig. 7. Hence, phase approval becomes a sub-process `Phase_Approval`. Again, applying Rule 7, we see in that `Project_Team_Setup` is a point of return. Thus, we omit the sub-process `Phase_Approval` in the description of `Director` and describe the role Director in Fig. 7 using the sub-process, `Project_Team_Setup`.

Therefore, we do not have `phase-approval` as an event in the alphabet of `Director`.

Instead the alphabet of the sub-process Project_Team_Setup gets added to its alphabet.

αProject_Team_Setup := {discuss, approve, abandon, approve-rework}

The alphabet of Director thus becomes:

αDirector := αDirector $\cup \alpha$Project_Team_Setup.

Similarly considering the roles Project Leader, Marketing Rep and Engineering Rep we can describe them as,

Project_Leader := assign-team → Project_Team_Setup

Marketing_Rep := assign-team → Project_Team_Setup

Engineering_Rep := assign-team → Project_Team_Setup

The alphabets of Project_Leader, Marketing_Rep and Engineering_Rep becomes:

αProject_Leader := αProject_Leader $\cup \alpha$Project_Team_Setup

αMarketing_Rep := αMarketing_Rep $\cup \alpha$Project_Team_Setup

αEngineering_Rep := αEngineering_Rep $\cup \alpha$Project_Team_Setup.

Although the state project-team-setup is common to the roles Director, Project Leader, Marketing Rep and Engineering Rep, the behaviour of each role after that state differs. That is, the starting points (initial states) are the same for each role but the subsequent triggers and sub-states are not. Hence, using Rule 11 we get:

Director := approval → (assign-team → Director_Project_Team_Setup)

Project_Leader := assign-team → Project_Leader_Project_Team_Setup

Marketing_Rep := assign-team → Marketing_Rep_Project_Team_Setup

Engineering_Rep := assign-team → Engineering_Rep_Project_Team_Setup.

(4 & 5) Fig. 6, has no choice or parallel constructs, therefore we will consider interactions (Rule 6).

(6) The interaction discuss is an event shared by the processes Customer and Customer_Contact. The interaction approval is an event shared by the processes Customer_Contact and Director. The interaction assign-team is an event shared by the processes Director, Project_Leader, Marketing_Rep and Engineering_Rep. The interaction produce-phase-plan is an event shared by Project_ Leader, Marketing_Rep, and Engineering_Rep. As described before, the interaction phase-approval (shared by Director, Project Leader, Marketing Rep and Engineering Rep) becomes a sub-process. In the process Director alone this sub-process will be referred to as Project_ Team_Setup but, in Project_ Leader, Marketing_ Rep, and Engineering_Rep the reference will be as phase-approval (qualified by the role name). Putting this all together we get the following CSP.

```
Customer := discussion → SKIP

Customer_Contact := discussion
  → (approval → SKIP)

Director := approval → (assign-team
  → Director_Project_Team_Setup)

Project_Leader := assign-team
  → Project_Leader_Project_Team_Setup

Marketing_Rep := assign-team
  → Marketing_Rep_Project_Team_Setup

Engineering_Rep := assign-team
  → Engineering_Rep_Project_Team_Setup

Project_Leader_Project_Team_Setup
  := produce-phase-plan
      → Project_Leader_Phase_Approval

Marketing_Rep_Project_Team_Setup
  := produce-phase-plan
      → Marketing_Rep_Phase_Approval

Engineering_Rep_Project_Team_Setup
  := produce-phase-plan
      → Engineering_Rep_Phase_Approval
```

The RAD in Fig. 7 describes the interaction 'phase-approval' between the roles Director, Project Leader, Marketing Rep and the Engineering Rep and, hence, is a part of the RAD described in Fig. 6.

The sub-processes which represent the interactions: phase-approval, Director_Project_Team_Setup,

Project_Leader_Phase_Approval, Marketing_Rep_Phase_Approval, and Engineering_Rep_Phase_Approval are described below.

Applying our rules to Fig. 6 and Fig. 7 we get the following CSP.

```
Director_Project_Team_Setup :=   discuss → (
                                        (Approve; Feasibility_Director)
                                        |      Director_Not_Approved
                                           )

Director_Not_Approved :=   Abandoned
                         | Director_Approved_Rework

Abandoned    :=   abandon → SKIP
Approved     :=   approve → SKIP

Director_Approved_Rework   :=  approve-rework →
Director_Project_Team_Setup

Project_Leader_Phase_Approval :=   discuss →     (
                                   (Approve; Feasibility_Project_Leader)
                                   |       Project_Leader_Not_Approved
                                                  )

Project_Leader_Not_Approved  :=    Abandoned
                                 | Project_Leader_Approved_Rework

Project_Leader_Approved_Rework :=  approve-rework → Project_Leader_Project_Team_Setup

Marketing_Rep_Phase_Approval :=   discuss →      (
                                  (Approve; Feasibility_Marketing_Rep
                                  |      Marketing_Rep_Not_Approved
                                                  )

Marketing_Rep_Not_Approved  :=     Abandoned
                                 | Marketing_Rep_Project_Team_Setup

Engineering_Rep_Phase_Approval := discuss →
            (Approve; (Feasibility_Engineering_Rep||Project_Support_Office))
            |      Engineering_Rep_Not_Approved

Engineering_Rep_Not_Approved  := Abandoned
                               | Engineering_Rep_Project_Team_Setup
```

Again applying our rules to Fig. 8, we get the following CSP (note that we have omitted the alphabets for brevity). Taking the role Director into consideration in Fig. 8 we get:

```
Feasibility_Director
   := Feasibility_Director_Launch_Review
```

where Feasibility_Director_Launch_Review describes the details of the complex interaction launch-review and will be described under Fig. 9. Since Feasibility_Director is a point of return, applying Rule 7 to the above CSP, we ignore Feasibility_Director_Launch_Review and describe the interaction (Fig. 9) under the name, Feasibility_Director.

Taking the remainder of the roles into consideration we get the CSP description given below.

```
Feasibility_Project_Leader := initiate-document-production →
                                    (Feasibility_Project_Leader_Describing;
                                     Project_Leader_Launch_Review)

Feasibility_Project_Leader_Describing := review →
                                   (Feasibility_Project_Leader_Describing
                                    | Proposal_OK)

Proposal_OK :=          merge-plans-and-proposals-for-review → SKIP


Feasibility_Marketing_Rep :=   initiate-document-production →
                                    (Feasibility_Marketing_Rep_Describing;
                                     Marketing_Rep_Launch_Review)

Feasibility_Marketing_Rep_Describing := produce-product-proposal →
                                  (review →
                                        (Feasibility_Marketing_Rep_Describing
                                        | (Proposal_OK)
                                   )

Feasibility_Engineering_Rep := initiate-document-production →
                               (Docs_Production :
                                Engineering_Rep_Launch_Review)

Docs_Production :=    (Feasibility_Engineering_Rep_Describing || Planning) :
                      ProposalOK_and_PlanMerged)

Feasibility_Engineering_Rep_Describing :=
       produce-product-proposal → (review →
                                    (Feasibility_Engineering_Rep_Describing
                                    |SKIP)
                                  )

Planning :=     produce-project-plan →
                           (check-plan → (Planning
                                         |Engineering_Rep_Plan_Checked))

ProposalOK_and_PlanMerged     := merge-plans-and-proposals-for-review → SKIP

Engineering_Rep_Plan_Checked :=  merge-proj-plan → SKIP

Project_Support_Office :=    Check-plan →
                                  (   Project_Support_Office
                                    | Project_Support_Office_Plan_Checked)

Project_Support_Office_Plan_Checked  := merge-proj-plan → SKIP
```

The interaction 'launch review' among the roles Director, Project Leader, Marketing Rep and the Engineering Rep is elaborated in Fig. 9.

The RAD given in Fig. 9 represents the CSP processes: `Feasibility_Director`, `Project_Leader_Launch_Review`, `Marketing_Rep_Launch_Review`, and `Engineering_Rep_Launch_Review`.

```
Feasibility_Director := discuss → Launched
                                | Feasibility_Director_NotApproved
Launched := launch → SKIP

Feasibility_Director_NotApproved := Feasibility_Abandoned
                                  | Feasibility_Director

Feasibility_Abandoned := feasibility-abandoned → SKIP

Project_Leader_Launch_Review := discuss →
                                ( Launched
                                | Feasibility_Project_Leader_NotApproved)

Feasibility_Project_Leader_NotApproved := Feasibility_Abandoned
                                        | Feasibility_Project_Leader_Approved

Feasibility_Project_Leader_Approved := Feasibility_Project_Leader_Initiate
                                     | Feasibility_Project_Leader_HandOver

Feasibility_Project_Leader_Initiate := initiate_documentation → Feasibility_Project_Leader_Describing

Feasibility_Project_Leader_HandOver := hand-over → Feasibility_Project_Leader

Marketing_Rep_Launch_Review := discuss →
                               ( Launched
                               | Feasibility_Marketing_Rep_NotApproved)

Feasibility_Marketing_Rep_NotApproved :=
                                 Feasibility_Abandoned
                               | Feasibility_Marketing_Rep_Approved

Feasibility_Marketing_Rep_Approved := Feasibility_Marketing_Rep_Initiate
                                    | Feasibility_Marketing_Rep_HandOver

Feasibility_Marketing_Rep_Initiate := initiate_documentation →
                                      Feasibility_Marketing_Rep_Describing

Feasibility_Marketing_Rep_HandOver := hand-over → Feasibility_Marketing_Rep

Engineering_Rep_Launch_Review := discuss →
                                 ( Launched
                                 |
Feasibility_Engineering_Rep_NotApproved)

Feasibility_Engineering_Rep_NotApproved :=
                                   Feasibility_Abandoned
                                 | Feasibility_Engineering_Rep_Approved

Feasibility_Engineering_Rep_Approved := Feasibility_Engineering_Rep_Initiate
                                      | Feasibility_Engineering_Rep_HandOver

Feasibility_Engineering_Rep_Initiate := initiate_documentation → Feasibility_Engineering_Rep_Describing

Feasibility_Engineering_Rep_HandOver := hand-over → Feasibility_Engineering_Rep
```

4.5. Mapping between RADs and CSP: summary

We have demonstrated how we can apply mapping rules in order to move from a RAD description of a process to CSP. We can then test the logic of our process, by stepping through a CSP description of the process using Enact as the process modelling engine. This enables us to reason about processes in a more formal way, and to discover problems with process that merely drawing pictures of the process would not discover. Indeed, in our own work, we have found the RAD to CSP mapping itself to be an iterative process, with the mapping to CSP forcing us to rethink the original depiction of the process in RADs. Indeed, the mapping enables us to make changes in CSP, then go back and change the RAD, and to on round such a cycle until we reach a stable and agreed process description. This kind of process is vital to the process modeller. It enhances the understanding of the process, and adds value to the process modelling exercise.

The main advantage of simple diagrammatic modelling notations is that they are accessible to relative novices. This is particularly important in business process modelling because we often wish to check our model by exposing it to process users. (This is perhaps the main advantage that Role Activity Diagrams have over more formal notations like CSP). However, the relative ease with which the Role Activity Diagram can be understood by the novice user is paid for by the lack of formality. Contrast this with CSP, for which we have a stepper, which allows us to simulate the behaviour of the process. This suggests that there is benefit in attempting to use these two notations in a complementary way by using RADs to present and discuss the business process with process users, and then experimenting with the process with an equivalent CSP description.

5. Problems with our approach and further work in progress

The work described here is only a partial solution to our search for a coherent modelling method. It has four iterative phases:

- Describe processes using a notation which users understand (RADs).
- Map between the notation by applying rules.
- Experiment with processes using an executable notation.
- Present static understandable models of the process solution to users.

The two main problems with the approach, are that the mapping is time consuming, and that the final presentation notation is static. Hence, we aim to have executable notations which retain the ease of understandability of a notation like RADs, which can be used to present running (enactable) process solutions to non-technical users. Current work is developing the mapping idea further to produce modelling phases as follows.

- Describe processes using a diagrammatic notation which non-technical users will understand.
- Automatically generate enactable process code.
- Experiment with processes using an executable notation.
- Describe process scenarios to users with an executable model which is diagrammatic and easy to understand.

This new method uses a simple user-facing paradigm which is mapped in stages to produce a model based slide show, which provides such an understandable interface for presentation to users, but which is controlled by a rigorous process model. A prototype tool has been developed, and tried on portions of European business processes, and will be developed further as part of the PROCESS[3] project.

6. Conclusions

We believe that being able to map from one paradigm to the other gives a significant advantage to the process modeller. For example, it gives us a mechanism to test the logic of our RADs, and it gives us an alternative and accessible way to present CSP models to process users.

By combining notations, such that we can have mappings between notations like CSP and RADs we are much more likely to be able to gain greater insight about the nature of the process under study, and to narrow the gap between the business process and the IT which supports it. This paper supports this combination of paradigms by giving a mechanism for mapping RADs to CSP, and showing how this can be used for examples of process models.

We suggest that an effective modelling method is to use established user-facing models (in this case RADs) for process elicitation, and presentation, and to map these models to executable notations for more rigorous process experimentation. The work presented here provides a first step in this direction.

References

[1] A.M. Ould and C. Roberts, Modelling iteration in the software process. Proc. 3rd Int. Soft. Process Workshop. Breckenridge, Colorado, USA. 17–19 November 1986. IEEE Computer Society Press.

[3] http://dsse.ecs.soton.ac.uk/~ga/process.html.

[2] A.M. Ould, Business Processes Modelling and Analysis for Re-engineering and Improvement, John Wiley, 1995.
[3] C. Roberts, Modelling and Co-ordinating Change in Business Processes, 1993. Process Modelling Workshop for the BCS Bristol Branch, Bristol University, Co-Ordination Systems Limited.
[4] A.M. Ould, An Introduction to Process Modelling using RADs, illustrated by reference to the case study, 1992. IOPTCLUB Practical Process Modelling Mountbatten Hotel, Monmouth Street, Covent Garden, London.
[5] Co-Ordination (1994). RADitor version 1.5: Users Manual, Co-Ordination Systems.
[6] C.A.R. Hoare, Communicating Sequential Processes, Prentice-Hall, 1985.
[7] P. Henderson, The CSP stepper in enact — an executable specification, 1992, available as an ftp source from ftp.ecs.soton.ac.uk in pub/peter/various/csp.ps.
[8] P. Henderson, Object-Oriented Specification and Design with C + +, McGraw-Hill, 1993.
[9] D.J. Hatley and I.A. Pirbhai, Strategies for Real-Time System Specification. Dorset House Publishing, New York.
[10] D. Harel, On visual formalisms. Comm. ACM, 31 (1988) 514.
[11] B. Kramer, et al. Petri-net based models of software engineering processes. Proc. 23rd Annual Hawaii Int. Conf. on System Sciences, Hawaii.
[12] D. Miers, Use of tools and technology within a BPR initiative, in Business Process Re-engineering: Myth and Reality, 1994, Kogan Page.
[13] M.G. Hinchey and S.A. Jarvis, Concurrent Systems: Formal Development in CSP, McGraw-Hill, 1995.