POSD - a notation for presenting complex systems of processes

Peter Henderson

Department of Electronics and Computer Science
University of Southampton
Southampton, SO17 1BJ, UK
peter@ecs.soton.ac.uk

Graham D. Pratten

International Computers Ltd
Westfields, West Avenue
Kidsgrove, ST7 1TL, UK
G.D.Pratten@uk03.wins.icl.co.uk

Abstract

When trying to describe the behaviour of large systems, such as the business processes of large enterprises, we often adopt diagramming techniques based on derivatives of data flow diagrams. For very complex systems such diagramming techniques suffer from the inability to abstract uniformly from arbitrary subcollections of components. In this paper we present an extension to conventional diagramming techniques which solves this problem. We describe how we have applied this technique to some very complex business systems and illustrate its main points with a simple example. While we have used the notation to present models of business processes we conclude that it is applicable to the description of behaviour in any complex system of processes.

Background

We are concerned with the nature of change in large and complex computer-based systems. In particular we are concerned with distributed systems, comprising many individually complex, legacy components. Such systems have become the basis of all large commercial or industrial enterprises. But the nature of the business in which these enterprises engage is constantly changing. So it is necessary to change the supporting computer systems if these enterprises are to remain competitive. We conjecture that the right way to go about such changes, given the constraints imposed by the legacy systems, is to model the business process which the enterprise system supports and to show how this process maps onto the legacy components [[1]]. The model must be in a form which the owners of the business process can understand, so that the proposed changes can be properly discussed with them and so that the impact of alternative changes can be assessed by them. We take this need for business user involvement to imply that the model must be presented in diagrammatic form.

We have used many types of diagramming technique in our work. Data flow diagrams of the SSADM, SADT, IDEF or Petri Net variety are probably the simplest for business users to comprehend intuitively [[2]]. Consequently they are the kind of diagram we have made most use of over the years. Usually such diagramming notations use two types of component: *boxes* (typically) to denote processing and *lines* to denote data flow. The notation usually allows boxes to be nested, but no matter how deeply the hierarchy is formed, usually the items flowing between processes are at the same level of abstraction from the most detailed to the highest level diagram. For large, complex systems this proves to be a drawback

Over the last two years, along with colleagues, we have developed models of a number of very large businesses. For example we have modeled a significant part of the business process of a large financial organisation. Similarly, we have modelled actual and proposed schemes for the business process known as a loyalty scheme in a retail organisation. Reports on these and other models are available on our Web pages [[3]]. The most detailed models were indeed data flow models. But abstractions from them were presented in a new form which we have called POSD diagrams (for Process Oriented System Design). Figure 1 shows a POSD diagram of a part of the business system we have modeled for the retail sector. This will be described in the next section when POSD notation is discussed.

Each of these business systems is modeled at many levels of abstraction and thus it is possible to show the mapping between levels. Also, different views of the same system are constructed. In particular we construct the low level view where the basic components map exactly on to services provided by the distributed support system. One can abstract from that in different ways, making different high level views. Abstractions are formed by combining components into higher level collections based on the structure of the distributed system or based on the structure of the component business processes. These two views in particular allow one to judge the effect of proposed business process changes in terms of the

To be presented at the First IEEE International Conference on ENGINEERING OF COMPLEX COMPUTER SYSTEMS, Florida, USA, November 6-10, 1995

changes required to the legacy systems.

POSD Notation

The name we have given the notation reflects our current use for it. We are using POSD (Process Oriented System Design) notation to model the business systems of large end-user organizations. We will normally model the existing business processes and, because proposed changes are the driving force, we will then model alternative future reorganizations of the business process. By showing the mapping to the installed legacy computer systems we can discuss the cost/benefit of each proposal with the owners of the business process. POSD is always used in conjunction with some base-level modelling language, such as data flow diagrams. Initially we work bottom-up. Typically, early attempts at modelling a business process are at (or only a little above) the data flow (document flow, work flow) level. After a while our understanding of the model is such that we can begin to form more abstract views. The principal way in which this happens is by a process of abstraction from the low-level

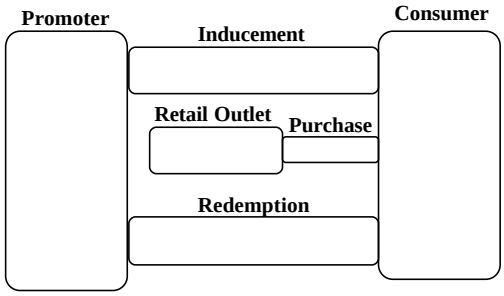

Figure 1 POSD model of Retail loyalty scheme

data flow diagram. Logically related components are combined into more abstract entities, and eventually a hierarchical view is arrived at bottom-up. It is this process of abstraction which the data flow diagram (and its relations) does not support sufficiently well. What we needed was a notation which allowed arbitrary subcollections of components to be combined to form a single new, more abstract component.

The consequence of this observation is that, whatever the nature of a collection of components, there must be an abstract component of a suitable "type" to which we can abstract that collection. The simplest solution to this apparent dilemma is to have only a single type of component. Since we a primarily concerned with process modelling we have termed our single component type a "behaviour". A behaviour is a component which has state, performs internal actions and interacts with other behaviours. Elsewhere we have given a more comprehensive description of this concept [[3]]. Clearly we can see that the usual process/activity element of a data flow diagram is a kind of behaviour. So too are the usual data repository components. With a little more thought, so too are the various means of data transfer among components.

Consider the simple POSD model shown in Figure 1. This comes from our model of a business process in Retail where the Consumer will Purchase items from a Retail Outlet. The fact that the Consumer's interface with the Retail Outlet is a Purchase is denoted by the juxtaposition of the boxes, by the fact that they actually touch each other. Each box is a behaviour. This diagram is an abstract view of the loyalty scheme. At this level of detail we have not said whether the method of purchasing is direct or by mail order or any of a wide variety of schemes. There are other relationships shown in the diagram. A Promoter encourages the Consumer to make certain purchases by offering Inducements (typically the promise of a gift or a discount). The Consumer can subsequently redeem this inducement, apparently by an interaction (called Redemption) with the Promoter. This particular abstract model is only one of many views of the Retail business process derived from lower-level data flow diagrams. Some views are organised (as this one) to show the business oriented abstractions while others are organised to show the location of the distributed systems which support this process.

Each box in a POSD diagram is a behaviour. Sometimes we show two levels on a single diagram. When we do, the behaviour of an outer box is implemented by the combined behaviours of the inner boxes. If two boxes touch this implies that there is direct interaction between them. If two boxes do not touch this implies there is no direct interaction between them. We have not restricted interaction to data (or other artifact) flow. Usually we refer to this interaction as *shared behaviour*, for a reason which will soon be apparent. Since behaviours will be made up of component behaviours we will expect interaction between touching behaviours at one level to be realized in some way at the more detailed level. We refer to the fact that two behaviours touch as a *promise* that we will (in a more detailed diagram) define how that interaction is accomplished.

There are basically three ways in which it can be accomplished as shown in Figure 2. A and B are behaviours which interact. This interaction is accomplished either by the fact that

1. each of A and B contains sub-behaviours which interact at the lower level, (here C in A interacts with D in B),

or

2. there is a shared sub-behaviour which is common to both A and B, (here C is common) or finally

3. one of A or B contains a sub-behaviour which interacts with the other high level behaviour, (here A contains C which interacts with B).

Clearly all promises are ultimately resolved by an application of rule 2. It is more normal to draw the two halves of the diagram for rule 2 separately and denote the fact that A and B share behaviour C by the fact that the common sub-component has the same name in each half. Sometimes we will draw the two halves overlapping, but our experience is that this not only leads to cluttered diagrams, it leads to confusion.

These are all the core diagramming concepts of POSD. POSD is intended for use with a suitable base-level modelling notation. We have used it mostly with DFD's, with Role Interaction Diagrams, with Finite State Machine notations and with Petri Nets. The only other POSD concept which needs to be described is how component

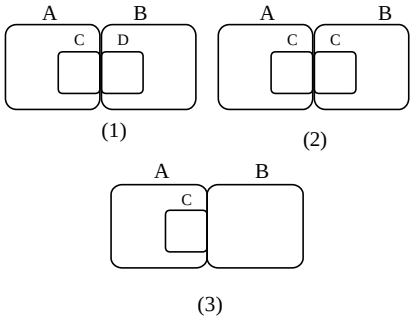

Figure 2 Ways in which promises are fulfilled

behaviours are named. So far, in our examples, we have used global names for instances of behaviours and in general this will serve. But for large systems, names become an issue and in the next section we will discuss this issue in the context of a simple example.

A Simple Example

We have chosen to detail the interaction between a Person and a Mail System as shown in Figure 3. Here we show that the Person and the Mail System each contain sub-behaviours which we have called Send and Receive. The interaction between a Person and a Mail System is fulfilled by the interaction between a Send and a Receive. It is necessary to distinguish between the type of a behaviour and an instance of a behaviour of that type. In the example we have show three instances of the Send

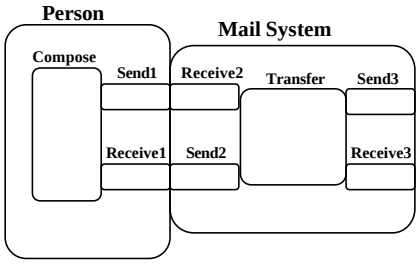

Figure 3 A Person's interaction with the Mail

behaviour and distinguished them by indexes.

In general, we are not pedantic about names on diagrams. But we have developed a reference model for POSD in which a naming scheme is defined (see papers on our Web pages [[3]]) where the distinction between types and instances is clearly denoted. For example the instances of the Send behaviour would be denoted fully as Person.Send1, MailSystem.Send2 and MailSystem.Send3. It is not intended that such a naming scheme is used by modellers but rather that full names are derived by the tools which support POSD diagramming.

At the next level of detail in our example, we have to show how the Send and Receive behaviours interact. We have supposed that this model is *bottomed-out* as a Petri-Net. Some of the detail is shown in Figure 4. Here we have a shared behaviour T common to both Send and Receive. A formal model should be accompanied by the equation Send.T=Receive.T but in practice we allow the fact that the name T is common to denote this equality.

Of course the purpose of these models is for us to be able to confirm that we have made an accurate presentation of the system and for us to be able to discuss the system with the owners of the business process. It is most likely that we would show only the top-level most abstract models to the business process owner, reserving the more detailed (and demanding) models to our own technical uses. Nonetheless, the lower level models are vital to an

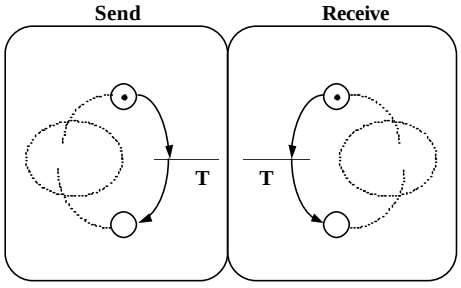

Figure 4 Detail of interaction between a Send and a Receive

accurate presentation of the abstract models and, as we said earlier, are often the way-in to the modelling project. This is because it is often easier to get to grips with detail to begin with. When abstraction comes, it is of course the key to a clear presentation, which is in turn the key to clear understanding. And understanding must precede cost effective change.

Conclusions

The method we have presented has served us well in a number of large scale system analysis activities performed by ICL. In particular, models of the business processes of a Financial Services business, of a Retail business and most recently of a News Agency business have been built on behalf of customers. The analysis has involved traditional tools and techniques complemented by POSD. In particular we have used the Process Wise process modelling tools which are a commercial product based on the IPSE 2.5 research prototype ([[4]], [[7]], see also [[6]]) to capture many of the base level processes. Process Wise has been extended to include POSD diagramming capability. We plan further tools, in particular a database/configuration management tool which will allow the organization and reorganization of a large set of inter-related diagrams. In particular this tool will keep track of unfulfilled promises. A prototype has been built using Access.

During the POSD analysis process, as with any systems analysis process, many diagrams are generated which are eventually superseded by improved diagrams. But in POSD, we also need to maintain simultaneously many different views (abstractions) of the same base process. The importance of documentation in the engineering of large complex systems is well established [[5]]. Consequently, a tool to facilitate the production of different presentations is urgently needed.

POSD is in the later stages of definition. It is being case hardened. The concepts presented here are just the core concepts. In practice it is necessary to adopt naming conventions for components which mirror the customer's usual conventions. As we have said, we have methods of naming types and instances of behaviours which satisfy this need. Also in practice it can be useful to use diagramming conventions different from those used here. For a long time we used both boxes and lines (which we tried to think of as shriveled boxes) to overcome some predictable topological problems (e.g. 5 boxes all of which need to touch each other). Lately we have dropped this extension ourselves because we believe it encourages us to accept *still-too-complex* models. But we are not certain that we won't return to it and we don't discourage it in others. There are some types of complex interaction which require the simultaneous participation of more than two behaviours and this is not well represented by juxtaposition.

POSD has been presented as a process modelling method, specifically for business processes. But we believe that it can be used more widely. Indeed for anything that data flow diagrams or finite state models or indeed Petri Nets have been used for, both hardware and software. We are planning such applications ourselves and hope soon also to publish our formal reference model with which we can confirm some of our conjectures about the applicability of the ideas.

References

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

[2] John Buxton and John McDermid *Architectural Design* in Software Engineer's Reference Book, Butterworth Heinemann, 1991

[3] Peter Henderson and Graham D Pratten *POSD - Process Oriented System Design*, CS/IN/2296, International Computers Ltd, February 1995,accessible, along with other POSD reports, from http://louis.ecs.soton.ac.uk/~ph/cv.html

[4] Robert A Snowdon *An introduction to the IPSE 2.5 Project* ICL Technical Journal 6(3), 1989

[5] David L Parnas *Software Aging* Proceedings of the 16th International Conference on Software Engineering, IEEE Computer Society Press, 1992

[6] Ian Robertson *An Implementation of the ISPW-6 Process Example* in Software Process Technology, Proceedings of EWSPT 94, Springer Verlag, LNCS 772, 1994

[7] Brian C Warboys *The IPSE 2.5 Project: Process Modelling as the basis for a support environment* in Proceedings of the First International Conference on Software Development Environments and Factories, Berlin 1989