
Incremental process deployment

Martyn A Ould
Venice Consulting Ltd
The Old School, Hinton Charterhouse,
Bath BA2 7TJ, UK

phone +44 (0)1225 723 822
fax +44 (0)870 056 7443
e-mail mao@veniceconsulting.co.uk



I want to talk about a couple of strands of activity which started from the same source and have once more converged.

First, a little history. Way back in 1986, Clive Roberts and I designed an object-oriented process modelling language, PML, which has some important properties. Firstly, it had to be executable, in other words it had to be possible to describe a process using PML, to give that description to a machine, and to have that machine enact or execute the process. We provided PML with a complete set of operational semantics in terms of first order predicate calculus with time, thus allowing us to describe the temporal behaviour of a process. The object-oriented property of the language was of course important because, through instantiation, we had the mechanism for creating new instances of the object types, in other words for making things happen.

I do not want to make this a tutorial on PML but it is appropriate for this workshop to hint at how the language captured process logic, in particular, not least because I believe the concepts remain valid and they now are proving fully amenable to direct implementation using current technologies. I shall gloss over all subtleties in the cause of brevity.

PML was built on an earlier language RML. It has *role*, *activity*, *assertion*, and *entity* as its primitive object types. Each object class is defined through *definitional properties* grouped in property categories. So, to define a particular role, one lists the definitional properties of its class within the property categories. When an instance of the role is created, it is defined by *factual properties*, which take instances of other classes as their values. Here are the property categories. (An early version of PML is defined in [1].)

Property categories of ACTIVITY

- /// **input** an ENTITY participating in the ACTIVITY and taken from the given property value class
- /// **output** an ENTITY inserted into the given property value class by this activity
- /// **control** an ENTITY participating in the ACTIVITY but not removed from the given property value class by this ACTIVITY
- /// **precond** an ASSERTION that must be true at the start time
- /// **postcond** an ASSERTION that must be true at the end time
- /// **actcond** an ASSERTION which, if it becomes true at any point, causes an associated instance of the ACTIVITY to begin at that point
- /// **stopcond** an ASSERTION which, if it becomes true, causes the ACTIVITY instance to stop at that point
- /// **part** an ACTIVITY that must occur in order for the instance to occur

Property categories of ROLE

- /// **association** an associated ENTITY (eg actor or resource) or ACTIVITY that might change over the life of the instance
- /// **control** an ENTITY participating in the ROLE but not removed from the given property value class by this ROLE
- /// **invariant** an ASSERTION that is always true of the instance

Property categories of ASSERTION

- /// **argument** an ENTITY that is an argument of the ASSERTION
- /// **neccond** an ASSERTION that is a necessary condition for the instance to be true
- /// **suffcond** an ASSERTION that is a sufficient condition for the instance to be true
- /// **defn** an ASSERTION class every instance of which gives a **neccond**
- /// **constraint** an ASSERTION expression with a non-null value

Property categories of ENTITY

- /// **producer** an ACTIVITY that produces the instance
- /// **consumer** an ACTIVITY that consumes the instance
- /// **modifier** an ACTIVITY that modifies the instance
- /// **part** a non-null ENTITY that is a component of an instance and that is fixed for the lifetime of the instance
- /// **association** an associated ENTITY that might change over the life of the instance
- /// **necpart** a **part** whose value is never null, ie that must always be present
- /// **invariant** an ASSERTION that is always true of the instance
- /// **initcond** an ASSERTION that is true when an instance comes into being
- /// **finalcond** an ASSERTION true at the time an instance ceases to be

A process model in PML is no more therefore than a pile of class definitions. The execution engine itself appears as a role called 42 containing activities that will allow users to create and instantiate classes. What you give to the engine is that pile of class definitions, including 42. The engine essentially constantly looks for instances of assertion classes. They represent, in

particular, the activating conditions of activities and so the process starts and runs. Instances are created. Things happen. Stuff is made.

The second property of PML, or rather of its execution engine, was that of *lazy instantiation*. One of the main requirements of this executable language was that it should be possible to change a process while it was running on the execution engine. It would not be acceptable to have to hold up existing processes or interrupt the flow of activity on the engine in order to change a process that it was running. We realised that this meant, in particular, that the engine should never instantiate any class ahead of time, everything had to be done only at the instant it was required. So, if a process was halfway through execution, only active or past instances of roles, and activities, and interactions, and the various entities, would be found in the engine. If we were then to change the class defining the process, the whole thing would move forward in the new altered direction, and there would be nothing which would be invalidated by that change. This had major implications for the notion of instantiating a process, for instance. The operational semantics supported these requirements.

As well as the formal language PML, formal in the sense that it had a fully worked-out set of semantics, we also provided a relatively informal language, the *role activity diagram*. (The use of role activity diagrams is described in [2], but significant improvements have been made since that book.) There was a simple set of rules for transforming a RAD into PML. Of course, a process drawn as a RAD, if drawn too much like a flow chart, can be made a very inflexible process, and we recognised that real processes would need to be rather loose fitting, in particular allowing complex circumstances to cause separate but communicating threads of activity. That loose fit can be quite adequately achieved with a RAD, but, to get from a rigidly modelled process to a more realistic elastic process, we also had a mechanism which we called 'relaxing the model'.

Clive Roberts and a small team built a prototype execution engine that ran PML models. What strikes me now is the inadequacy of the tools that they had at their disposal in 1986. It was written in SmallTalk, but it worked.

My involvement with the concepts stopped about then.

Six years later, the TQM and BPR industries were gathering pace, and it occurred to us that we had some useful approaches to thinking about processes somewhere on a dusty shelf. Nobody of course was very much interested in an executable object-oriented language with lazy instantiation. But role activity diagrams looked as though they had some potential, so we dusted them off, and started developing a method around what is of course just a notation. The method was called STRIM, a name I never liked. I guess that, on and off, since then I have been developing that method further, now under the name *Riva*. A central and important feature of the method re-

mains: that of treating organisational activity as essentially a collaborative affair with a business goal in mind. In particular, we divide activity up amongst roles (typically areas of responsibility or posts) and we design interactions between those roles to coordinate them.

One of the important developments that I have made to the method is the concept of the 'process architecture'. I have to say that we never had a satisfactory answer in STRIM to the question 'what *are* the processes in this organisation?' Or to the question 'how do I handle a *large* process?' The rest of the world seemed happy with the fiction of hierarchical decomposition, but I never allowed the concept in STRIM. It was only after a number of situations where actually we dealt with seriously large processes such as the development of pharmaceutical drugs that I understood how the process architecture could and should be, mechanically as it turns out, derived from an understanding of the subject matter of the organisation concerned, what I call the *essential business entities*.

The major advantage of a process architecture developed with **Riva** (as outlined in [3]) is that it is a constant for an organisation that stays in the same business. If the organisation changes the business that it is in, by perhaps moving into new areas or outsourcing existing areas, then corresponding changes in the subject matter of the organisation cause well-defined corresponding changes in the process architecture. A **Riva** process architecture proves to provide a very reliable chunking method, cleaving the organisational activity along natural fault lines, lines that are quite independent of functional structures or culture or any other organisational design decision. In particular it avoids the assumption-laden mistakes that people generally make when they decide on what the organisation's processes are by drawing some sort of sequence of major steps (what I call kebab modelling) or by taking the names of functional groups and turning them into process names. In any work that I do now with clients I always insist on a process architecture as the first step even if we are only looking at one process, and they think they know what that process is, what it is called, and where it starts and ends.

So what about the convergence that I mentioned at the beginning? Well, Clive Roberts has in recent days been building a new process engine, called InterActor. He has returned to the original ideas published in 1986 but now of course he has a very much richer set of technologies at his disposal, with a great deal of off-the-shelf software which brings many things for free which had to be built by hand 17 years ago. He and I and another colleague David Perrin have been developing a joint approach using **Riva** as the method and InterActor as the engine.

Following our original work, the role and the interaction are central to InterActor. InterActor is a coordination machine. Roles can be defined as

classes comprised of activities and interactions with other roles. Those roles can then be instantiated, as the process proceeds, and the role instances can coordinate – collaborate – through interaction instances.

Another of the important engine properties that Clive has carried forward in his work is the ability to change the process on-the-fly. If we combine that with **Riva**'s process architecture concepts we can address the vital requirement of BPM of supporting the *agile* business – something Howard Smith has reminded us of in his *Third Wave* book – in particular where the agility is manifested in the workplace rather than in the IT Department. **Riva**'s role activity diagram remains the principal view of the process with which end-users interact. (The editing process is of course part of the process held by InterActor.) This leads us on to what we refer to as 'grow-as-you-go'. We believe that it will be important for an organisation to be able to deploy an InterActor-based solution in a small area in an experimental way, to grow that deployment to include more process and/or more organisation, and to allow the process to be developed by the business, so that indeed the process is the business. It's for this reason that I prefer, in a process enactment context at least, to talk about the *process potential* rather than the process model. The thing in the engine determines what – as of this moment – could happen in future, in the same way that the petrol in a car engine contains the latent energy that will become kinetic energy when the engine consumes it.

And there is another thing in the process engine – which, if I were to continue my analogy, would include the exhaust gases – which a petrol engine does not hold: the results of execution, namely past instances of roles, activities, interactions, and entities. If it retains these, if it has *total persistence*, I am assured of being able to provide the end-user with everything in context. Our end-users can live in the engine. They need nothing outside the engine. In particular, they do not need dreary filing systems, databases, or document management systems to keep their stuff. Such monsters extract the context from things and leave us needing to remember where we put them, or what we called them, or what relationships they have to other things we know about. Thanks to persistence, our process enactment system will be able to present the *process past* and the *process potential* – everything – in context. And we can dispense with all those secondary storage systems.

It's useful here to raise another feature of **Riva**, to do with the way that the process architecture is constructed. I mentioned that the architecture is derived mechanically from the essential business entities. In fact, we filter those entities to find the ones which represent what we call 'units of work', an obvious example for many organisations being the customer.

We then assert that each unit of work has associated with it three processes in the organisation: the *case process* which deals with a single instance of the unit of work, the *case management process* which deals with the flow of instances of the unit of work, and the *case strategy process* which takes a strategic view of the unit of work and the other two processes for it. Our assertion is that the resulting process architecture covers all the activities of an organisation with those units of work. In other words, if I walk into the organisation's building and observe any business interaction or activity I will know that it lies in a process in the process architecture.

In a classical workflow situation, we would use **Riva** to determine the set of processes concerned and then to define in detail those that we wanted to deploy. In a grow-as-you-go BPM context things are different in one respect. We would still use **Riva** to establish the process architecture of the organisation and thereby chunk its activity properly. This will allow different parts of the organisation to develop their processes on InterActor simultaneously without fear of later finding themselves overlapping, conflicting, or generally getting in a twist, which we know that people do if they have not started from that first all-important chunking. That's **Riva** working at the architectural level. At the individual process level, it of course provides those end-users with a method for designing their collaborative processes in terms of real-world things such as roles, activities, and interactions, in the form of a RAD. The difference is that that definition will now be one that develops over time and with use, rather than being set in concrete at the outset.

Our experience in consultancy work has been that people need reliable ways of getting to grips with concurrency – with the parallelism in their organisational activity – with the fact that there are a zillion related things going on at one moment. They need to understand what concurrency there is, and they need to exploit the possibilities of increasing concurrency, in order to reduce cycle times. So-called process models that I see are too often serial affairs, which represent something that simply doesn't exist. We also need full mastery of concurrency in all its forms if we are to build real processes – big, ugly, complex, real processes – in an engine. With **Riva** we can achieve this at several levels. The **Riva** process architecture shows how concurrent processes interact, and which processes can have many concurrent instances and how they interact. The **Riva** RAD shows how concurrent roles interact, and which roles can have many concurrent instances and how they interact. And finally within a role, we can capture concurrent threads of activity, and which threads can have many concurrent instances and how they interact.

We have built a demonstrator, running on a network of three PCs, one being the server supporting the process engine and all three working as clients simply using a browser, and we were able to make simple changes to

the processes. Much now depends on the development of the next version of InterActor, currently in hand. A tool on its own is, of course, of little use unless one has a way of using it. Because **Riva** and InterActor derive from the same theory, we have a major advantage over tools that exist in isolation, perhaps simply built around just a notation. In **Riva** we have a method for thinking about processes using a particular underlying computational model, and in InterActor we have an engine that runs that computational model.

To demonstrate that nothing is new in this world, I'll quote Bob Snowdon from an article he wrote in a process technology journal that Praxis ran a decade ago called *IOPener*: '...there is clearly a variety of issues and problems outstanding. These range from the methodological (just exactly how do you develop process models and process model based support systems), through the technical (what should a process modelling language look like) and the ergonomic (how do you present participants with their view of what is going on) to the architectural and implementation (what capabilities are required of the underlying computer system to allow efficient and effective enactment, and how are tools incorporated).'

The questions remain valid, and we have some answers today.

But I would like to pick up one of those questions that, for me at least, is still unsolved: 'how do you present participants with their view of what is going on?' And I'd like to couch that in a different way: 'how many processes are there? in particular are there more than one?' A **Riva** process architecture chunks the organisation's processes into communicating case, case management, and case strategy processes. It's a way of reliably getting your arms round all the organisational activity. In fact, if you have N units of work then you have $3N$ processes. I assert that the list of those $3N$ processes is invariant, though we may choose to merge processes in that list, or to move boundaries here and there, for reasons of efficiency or simplicity. And of course, in a power-to-the-people sort of world, end-users will choose to do just that sort of thing on the fly. Where does that now leave that careful chunking? If I, an end-user, ask to see 'my process' so that I can inspect it and change it, what do I get shown? Do I get shown a chunk as defined by the process architecture, or do I get shown part of the potentially vast canvas on which the entire organisational activity – as it exists at this moment – is drawn. Moreover, do we not need some sort of chunking for change control, to prevent two of us from changing the same parts of the canvas at the same time? Put another way, are there limits to the freedom we can allow the end-users?

I suspect that we shall have to impose some sort of chunking, and if we are going to do that, then we had better impose a chunking that remains constant for at least as long as the organisation stays in the same business.

We would certainly not want it upset by a change to the organogram or a cultural change programme. A **Riva** process architecture satisfies those properties.

Let me summarise the vision we have for the **Riva** method and the InterActor engine with some catch phrases that we tend to use:

- ❖ **concurrency is the rule**: serialism is rare
- ❖ **organisations are collaboration**: processes are collaborative affairs, that collaboration spanning concurrency
- ❖ **it's all in the engine**: the process past and the process potential are all in the engine – it is the world, not a reflection of it
- ❖ **grow as you go**: processes rarely remain constant, so change is an everyday matter for InterActor
- ❖ **power to the people**: the changes that effect that growth can be made by the end-users
- ❖ **no more names**: everything being in the engine, in particular the process past, everything can be presented to the end-user in context.

References

- 1 'Defining formal models of the software development process', Martyn A Ould and Clive Roberts, in *Software Engineering Environments*, Ellis Horwood Limited, 1988
- 2 *Business Processes – modelling and analysis for re-engineering and improvement*, M A Ould, John Wiley, 1995
- 3 'Designing a Re-engineering-proof Process Architecture', Martyn A Ould, *Business Process Management Journal*, **3**, 3, 1997