# Reflection and Reification in Process System Evolution: experience and opportunity

R. Mark Greenwood[1], Dharini Balasubramaniam[2], Graham Kirby[2], Ken Mayes[1], Ron Morrison[2], Wykeen Seet[1], Brian Warboys[1], Evangelos Zirintsis[2]

1: {Department of Computer Science, The University of Manchester, Manchester M13 9PL, UK.}
2: {School of Computer Science, The University of St Andrews, St Andrews, Fife, KY16 9SS, UK.}

## Abstract

Process systems aim to support many people involved in many processes over a long period of time. They provide facilities for storing and manipulating processes in both the representation and enactment domains. This paper argues that process systems should support ongoing transformations between these domains, at any level of granularity. The notion of creating a enactment model instance from a representation is merely one restricted transformation. Especially when process evolution is considered the case for thinking in terms of model instances is weak. This argument is supported by our experience of the Process*Web* process system facilities for developing and evolving process models. The idea of hyper-code, which supports very general transformations between representation and enactment domains, is described. This offers the prospect of further improvements in this area.

## 1. Introduction

In the process modelling research community we often adopt a simplistic view that business, or software, process modellers aim to [Fei93, Dow94, Fin94]:

1. Model a process using a textual or diagrammatic representation. (This model is often called a process model definition. It is at a class or type level and static)
2. Translate (or compile) the model into some library form within a process support system (PSS).
3. Instantiate the library model to give a process model enactment. (This is often called a process model instance. It is dynamic and consists of some set of entity or object values within the PSS.)
4. The PSS runs the model enactment to support the process performance of a specific process instance.

There are of course design decisions about the important features of the process that we want to capture in our model, and about the proportion of the process that is to be supported by process technology.

The underlying assumption is that the textual or diagrammatic model represents current best practice, and one of the main benefits of modelling is to perpetrate this to all process performance instances.

In this paper we argue that this view is a special case. First, it is based on a single process class; we need to consider how a process support system will support many processes within an organization. Second, we should be thinking more generally about the relationship between the textual or diagrammatic representation, which we manipulate as process modellers, and the set of values within a process support system, which represent a specific model instance. In particular, we will argue that giving too much emphasis to the one-way translation from representation to entites/values places artificial limits on the potential of process support

systems. We describe the Hyper-Code technology [Zir00a,Zir00b] and illustrate how this promotes thinking about a more general two-way translation, which can be performed many times throughout the lifetime of the process.
Two-way translation is important when we consider the wider context of observing enactment state, perhaps to debug it, and reusing code and data within a PSS. We will illustrate our argument using details from the Process*Web* [PWeb, War99a] system with which we are most familiar. We believe that Process*Web* is a reasonable representative of the state of the art, and our argument does not depend on the specific details of how various facilities are provided in Process*Web*.

## 2. Process System Viewpoint

At any point in time a process support system (PSS) will include a set of process descriptions within its library. For each of these descriptions there may be zero or many process instances, each with their own specific state information. The PSS will also supply some general environment facilities to all models. For example, most process instances may involve multiple users and each user may be involved in multiple process instances, not all from the same description. As a user logging on and off the PSS is orthogonal to the progress of the process instances, the logging on and off facilities apply across all models. Indeed users may log onto the PSS when they are not involved in any process instances. The PSS must also provide facilities that allow users to navigate around the current descriptions and instances, and to bind users to specific roles or activities in a process instance.
We can think of a PSS as a virtual machine, or operating system, that runs process services. Each process enactment instance is a process service and has a corresponding description.

The Process*Web* system is based on the object-based language PML [PML]. This has four base classes: roles (an independent thread with local data and behaviour), actions (procedure invocations), entities (data structures) and interactions (buffers for transferring data between roles). In Process*Web*:

- Process descriptions take the form of a *set of PML classes*, and the name of a designated boot class. There is a library structure of these descriptions, indexed by their boot class name.
- Process instances consist of a network of PML role instances, linked by interaction instances. They are created by the built-in action *StartRole*. A unique name, provided at instantiation, is used to index each current process instance. (In Process*Web* the details of process instances are deleted when they complete, but they could be moved to a separate library of completed process instances.)
- The environment facilities are provided in two ways. There is a set of Process*Web* specific PML classes that can be referenced by name in the process descriptions. These can be used for stateless information: generic PML classes for the user interface, accessing an external Oracle database etc. There are also environment facilities that provide information specific to the current state of the PSS as a whole. In Process*Web* these are provided by 'management roles'. Whenever a process instance is created it is given a link (interaction) with one of these. This *Manager* role gives the process instance access to environment information through a set of request types to which it will reply.

The key point for our argument is that while process enactment instances are independent of each other, they exist in a common environment. We do not expect a separate PSS, with its own users, state etc., for each instance. Conceptually the PSS is a virtual machine that runs forever. Even without considering process instance evolution, which we will return to in section 6, the PSS as a whole evolves: for example, through the introduction of new process descriptions, of new users, and the creation of new instances.

## 3. Reflection

The issue that we skipped in section 2 is how we get from a textual or diagrammatic process representation to a description as a set of classes within Process*Web*. In Process*Web* the technique used is linguistic reflection, which is defined as:
"the ability of a running program to generate new program fragments and to integrate these into its own execution" [Mor00a] page 155.
The PML language includes a built-in action *Compile*. PML is a textual language and the PML source for a model is input to the PSS as a *String*. *Compile* takes as input this source string and an environment, which is a *set of PML classes*. It produces either an error message, of type *String*, or a new environment, which consists of the input environment plus the newly compiled classes.

In terms of linguistic reflection, the program is the Process*Web* system itself. The Process*Web* models are the new program fragments through which it extends itself. When Process*Web* is created, it includes in its description library a *Developer model*. This gives access to the *Compile* action, where the initial environment is the PML and Process*Web* standard classes.

For this paper, the key point is not the choice of linguistic reflection (or the detail of its implementation in Process*Web*), but that there must be some mechanism so users can incrementally add new process descriptions into a PSS. This will involve translating a model from the textual or diagrammatic representation used by a process modeller into the corresponding set of object/entity values within the PSS. In addition, there must be ways of establishing links with the generic environment provided by the PSS.

[We believe that linguistic reflection offers benefits in terms of the ability of Process*Web* to evolve itself as well as its process models while still running and providing a service to its users. However, this is outside the scope of this paper.]

## 4. Reification

It is no surprise to learn that not all Process*Web* models are perfect. Sometimes there are run-time programming errors. Sometimes the models just don't provide the expected process support for their users. When things go wrong a very important question is "What is the current state of this model instance?" The modeller needs a reification of the current state of the model, and possibly the PSS more generally.

If the current state is not deducible from the model's input, output and source text description then the modeller will usually resort to the diagnostic facilities. [The main advantage of the diagnostics is that they are interactive. There are programmable facilities for examining the current state, see section 6, but these are not as easy to use.] The Process*Web* diagnostic facilities provide text results in the general form:
*<variable name>* : *<type name>* = *<string representation of the value>*
Where the value is a PML object, e.g. an interaction instance or a role instance, then the value string will contain a unique identifier that can be given as a parameter to the relevant diagnostic command to provide additional information.

By issuing a series of diagnostic commands and noting the answers a modeller can build up a picture of the model instance's current state. However, the representation provided for the results from diagnostics is quite different from the PML source text representation. In essence, we are doing the inverse of instantiation. We start with a set of values within the PSS, which are the process instance, and transform them into a representation of the current

state for the modeller. When looking at the current state a modeller is often interested in a specific part of the model instance rather than the complete picture. It is often clear from the run-time error, or the observed problem, that it is a specific part of the problem that needs to be examined.

It is worth noting that errors do not have to be in the model instances. A specific model instance may expose errors in the general environment facilities. As more models are written new useful environment facilities that can be shared between models are discovered. In Process*Web* the diagnostic facilities can be used to give a representation of the state of the environment as well as the model instances.

## 5. Hyper-Code

This is a refinement of the concept of hyper-programming [Mor95], in persistent programming language research. In hyper-programming a program is no longer a linear sequence of text, but contains both text and links to objects in the environment that exist when the program is being constructed. This takes advantage of the fact that, in persistent systems, programs can be constructed and stored in the same environment as that in which they are executed. In traditional programs, where a program accesses another object during its execution, it contains a textual description of how to locate the object. This is resolved to establish a link to the object itself. Typically this is during linking for code objects, and during execution for data objects. The benefits of hyper-programming include:

- being able to perform program checking early: access path checking and type checking for linked components may be performed during program construction
- increased program succinctness: access path information, specifying how a component is located in the environment, may be omitted
- being able to enforce associations from executable programs to source programs
- support for source representations of all procedure closures: free variables in closures may be represented by links

Hyper-code takes these final two benefits a step further. The key idea in hyper-code [Zir00a, Zir00b] is that a single, uniform, program representation, the Hyper-Code Representation (HCR), is presented at all stages of the software process. The representation for creating new values and browsing existing values is the same. This maximizes simplicity for the programmer. It also means that there is no longer any distinction between an editor for creating new values, and an object browser for examining existing values. All programmer interaction with the system takes place through a single unified hyper-code editor. This blurs the distinction between compile time, bind time and run time.

We can think of a hyper-code system in terms of two domains, **E** (entities) and **R** (representations), and primitive operations between them. **E** is the domain of language entities: all the first class values defined by the programming language, together with various non-first class entities, such as types, classes and executable code. **R** is the domain of concrete representations of entities in domain **E**. A simple example is the integer value seven in **E** and its representation 7 in **R**. Domains **E** and **R** can both be divided into executable and non-executable subsets.

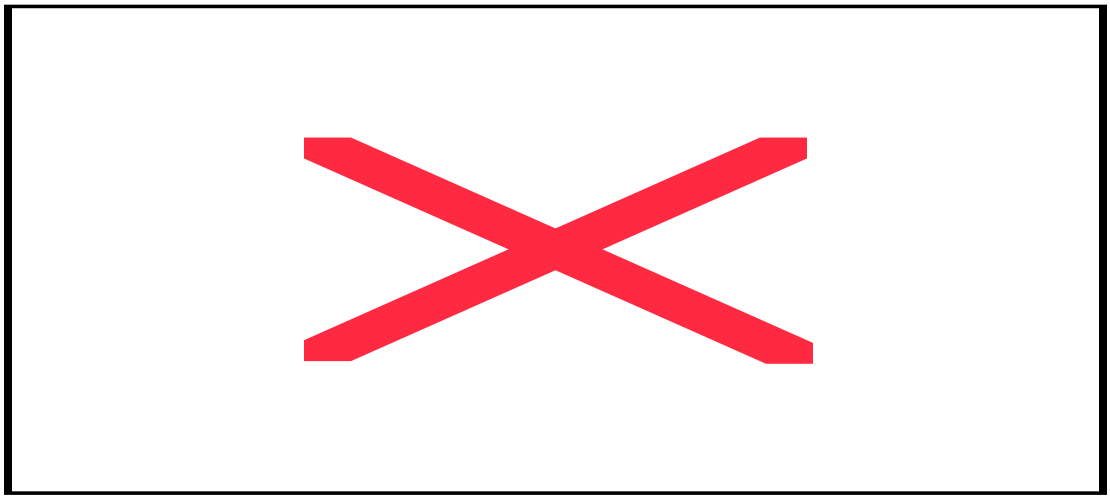

**Fig. 1** Domains and Domain Operations

The four domain operations (illustrated in Fig. 1) are:

- *reflect*: this maps a representation to a corresponding entity ( **R** ⇒ **E** )
- *reify*: this maps an entity to a corresponding representation ( **E** ⇒ **R** )
- *execute*: this executes an executable entity, potentially with side effects on the state of the entity domain. ( **E** ⇒ **E** )
- *transform*: this manipulates a representation to produce another representation ( **R** ⇒ **R** )

These abstract definitions may be interpreted in various ways for different concrete hyper-code systems. For example execute could involve strict, lazy or partial evaluation of an executable entity. The style of representation produced by reify could vary. The transform operation could be unconstrained, allowing any representation to be produced, or limit the possible representations as with a syntax-directed editor.

It is not intended that a user would access these domain operations directly. A user only interacts with the system through an editor that manipulates HCRs. [Zir00a] proposes a set of primitive hyper-code editor operations, based on the experience of building a hyper-code system for Java.

- Edit is essentially to transform.
- Evaluate takes r∈**R**, reflects r to the corresponding e∈**E**, and executes e. If this produces a result eres∈**E** then this is reified to rres∈**R**, and rres is displayed in the editor.
- Explode and Implode respectively expand and contract a selected HCR to show more or less detail, while retaining the identities of any values referred to. They both use reflect and reify.

Note that in a hyper-code system, during design, execution, and debugging, users only see HCRs. Entities such as object code, executable code, compilers, linkers, which are artifacts of how the program is stored and executed, are hidden from the programmer, since these are maintained and used by the underlying system.

We return to the simplistic view from section 1, and review it from a hyper-code viewpoint.

1. Model a process using a Hyper-Code Representation (HCR) - (*transform*)
2. Translate this into a Process Support System (PSS) - (*reflect*)
3. Instantiate ? - (no corresponding domain operation)
4. Run the model using the PSS - (*execute*)

There is a mismatch as there are no clear equivalents for instantiate and reify.

The purpose of instantiate is to get multiple process instances that follow the same definition. There are two ways this could be done without having instantiate built into the system. We could use transform to make a copy of a chosen representation and apply reflect to the copy, giving a distinct entity for execution. Alternatively, the representation could be not of a process instance, but of a process instance generator (constructor function). This could be executed many times; each execution having the side-effect of producing a new process instance within domain **E**.

As we have argued in section 4, reify is a familiar operation to those who develop and debug process models running on process support systems (although often only a limited form of reification is available). The fact that there is no corresponding transformation in the view, with which we started, is because of its simplified nature. There is an inter-relationship between instantiate and reify. Instantiation is often used as a way of keeping a reference between a model's source representation and its current enactment instances. However, if we can reify to a source representation there is no way that the source for an instance can be lost. If required some source representation comparison tool could be used to identify whether or not two enactment instances belong to the same class.

The concept of developing a process model, represented in some source format, and then creating multiple model enactment instances to support the process embodies connotations of timing and granularity. The assumption is that there is some specific creation time at which this occurs, and that it occurs for the whole model at this time. The hyper-code viewpoint is more general in that reflection and reification can occur throughout the life-time of entities, and because they can be applied at whatever level of granularity not just at the model instance level.

One conclusion is that instantiation is not a fundamental concept that needs to be primitive within a process environment. It may be very useful in many modelling contexts, a common process design pattern perhaps? This is reinforced by the fact that there are some processes, e.g. a business's budget allocation process, for which there should only be one instance within a PSS. In addition, it is very much a business modelling decision whether an insurance company wants to think about one instance per policy, or one instance per customer, or indeed one instance for the whole company.

There could be other benefits from reducing the emphasis on instances within the process modelling community. Martyn Ould remarks, "My (bitter) experience is that the effect on an ordinary person of hearing the word 'instantiate' for example is akin to a sharp blow between the eyes with a heavy club: it switches them off" [Oul95] page 27.

## 6. Evolution

Within process modelling thinking about ongoing transformations between source representations and their corresponding PSS entity values is common in the context of process evolution. The arguments for evolution in process systems are based on three main themes. First, organizations evolve their processes and the models of these must be kept up to date, otherwise process support systems hinder rather than enhance process performance. Second, organizations frequently have management processes for monitoring and evolving their operational processes, and these meta-processes can be modelled and supported. Third, many processes are themselves long-lasting and incrementally developed with later parts being dependent on results from earlier ones.

The simplest evolution for process systems to support is at the representation level. A modeller makes changes to the textual or graphical representation. This may then replace the changed process within the PSS library, or be made available through a new name or version identifier.

By allowing a lazy transformation from representations, a process system can support some evolution of a running model. Rather than have a single creation time instantiation of the complete process, the translation from representation is done incrementally. The initial parts of the process may proceed at the same time as the representations for later parts are being created and changed. Indeed, if the environment provides a process with facilities to access and update representations that it will use in future then earlier parts can create or modify the representations that will be used later. (This is analogous to many common situations, e.g. where part of a software development project is to define the testing procedures to be used by the project at a later date.) With this sort of approach there is a blurring of process design and process execution. If we think of instantiation it is of process model fragments, which need to be combined with existing process enactment fragments, rather than instantiation of complete process models.

Both of the evolutions described above are in a sense anticipated. In Process*Web* terms they do not involve any new facilities that have not been described previously. They do not cater for the case where a running model needs to be changed to take unexpected requirements (including working around run-time model errors).

The facilities in Process*Web* for unexpected run-time (instance) evolution are based around being able to change the code for a PML role instance. More extensive changes consist of a number or related role instance evolutions. The evolution is programmed in PML. In essence, one PML process is the meta-process that is evolving another. The basic sequence is based on several pre-defined PML actions:

1. *FreezeRole* is used to suspend the role instance
2. *ExtractRoleData* returns the current local data values (resources) of the role instance as a *table* indexed by the resource name
3. The new code for the role instance is created and compiled (*Compile*) into a *set of PML Classes*. If required the current source for the role instance can be extracted. *GetRoleClasses* is used to give the *set of PML Classes* and the role class name given when the role was started (see *StartRole* section 2). These can be passed to *GetClassDefinition* to get the role's current source as a PML String.
   The table of data values extracted from the role is updated so that it contains those required when the role re-starts according to its new definition.
4. *BehaveAs* is used to change the code of the role instance. Its parameters include the new set of PML Classes and the new role class name, and the table of initial data values. *BehaveAs* also restarts the role instance so that its execution will continue.

(There is a simpler sequence *BehaveAs*, *ExtractRoleData*, manipulate table of data values, *InsertRoleData* and *UnfreezeRole* that can be used if only the unexpected change of local data values is required.)

Collectively these PML actions provide a powerful set of evolution mechanisms that have been exploited in research into process model evolution and its support [Gre96, Gre00, Sa95, War99a, War99b]. They are however quite complex to use, and thus frequently the source of errors. When an error means that you need to evolve the meta-process, which you created to evolve another process because of its error, mistakes are easy to make.

PML is a textual language. This means that when writing the meta-process there is no way that the developer can refer directly, at development time, to the role-instances of the process being evolved. The binding between the meta-process and the process it changes is at

meta-process run-time. The new PML source for a role instance cannot refer directly to any of its current values. This is why the code and data parts have to be treated rather differently in Process*Web* evolution.

Using the hyper-code terminology, the meta-process, when it runs, must reify the evolving process into a representation that it can manipulate, and then reflect to make the evolution. The manipulatable representation for code is a PML String and for data a PML table mapping resource name to value. For code the reification is done by *GetRoleClasses* and *GetClassDefinition*, and part of the reflection is done by *Compile*. For data the reification is done by *ExtractRoleData*. The final part of the reflection is done by *BehaveAs*. This must check the consistency of the new code and the data to which it refers. For example if the code includes the definition of an Integer resource named "x" and in the data table "x" has a String value then there is an error. (In PML, *BehaveAs* offers two options: fail if the code and data do not match, or just discard the data binding and continue.)

The experience of Process*Web* illustrates that reflection and reification are at the heart of effective process evolution support. The restrictions imposed by the linear textual nature of the PML language are a major factor behind the complexity of the facilities required to evolve Process*Web* models. One final note about evolution in Process*Web* is that it is supported at the granularity of the role instance. While one role instance is being evolved, others conceptually in the same model can still be running.

Process evolution at the enactment or instance level is normally thought of in terms of modifying code. There is some process code written that needs to be changes. Any required data modifications are a secondary effect. We can look at this from an alternative perspective. Process evolution involves the current process code being suspended (and never restarted) yielding a set of entities within the PSS that represent the current state. It is replaced by new process code that refers to as much of that current state data as required. The key problem is how to refer to the current state data when writing the source of the new process code. Clearly, a facility to include hyper-links to existing data in the source makes this much easier. With a hyper-code approach it would be possible to be constructing the new process code at the same time as the old process code is executing, and thus minimize the changeover time between old and new process code. From this perspective there is not one overall process enactment that is evolved. In contrast the PSS runs a number of partial processes. It provides the facilities to suspend any partial process and start a new one referring to the data from the suspended partial process.

## 7. Conclusion

At EWSPT'94 Dowson and Fernstrom's paper [Dow94] discussed the idea of three domains: process definitions, process (definition) enactments, and process performance. Their diagram (page 92) includes one arrow from definitions to enactments. This is typical of the emphasis given to a single transformation from a representation domain (definitions) to an entity domain (enactments). We believe that a process support system needs to support the ongoing transformations between these domains. There is substantial benefit in thinking more generally about reflection, the transformation from representations to entities, and reification, the transformation from entities to representations. These are fundamental in a PSS. The one arrow in Dowson and Fernstrom's diagram is probably because their paper concentrates on the idea of separating the enactment and performance domains, and the relationship between these domains. They do later refer to the need for reification. "At the very least, a process manager making a change to an enactment needs to be able to inspect its current state, and tools supporting the presentation of views of enactment state are needed."

A hyper-code approach that would offer process modellers, using a PSS, a single source representation throughout the life of their model appears very promising. It means that developing, monitoring, debugging and changing can be interwoven as the modeller sees fit. The ability to link to existing entities within the PSS when constructing new hyper-code models would make re-use of model fragments easier. It would also greatly simplify the complexity the modeller currently faces when dealing with process evolution.

One issue to be resolved in how the primitive hyper-code domain operations should best be packaged and made available as concrete operations in a hyper-code process modeller. The concrete operations from the Java may not be the most appropriate. If we compare with Process*Web* there was a separation between suspending the thread associated with a role instance, and the reification to data structures that can be manipulated. Is this a separation that should be kept, should there be an option on the concrete operations that use reify? [Zir00a] reports some work on using a hyper-code representation (HCR) to trace the execution path of an entity during its evaluation.

This paper does not offer any measured benefits of the hyper-code approach. The importance of the underlying thinking certainly resonates with our experiences with the Process*Web* PSS. For example, the ability to reflect and reify at whatever granularity is appropriate, rather than just in terms of model instances. Our experience with Process*Web*'s facilities for evolution is that the problems of referring to existing data in new code is a major source of complexity.

The emphasis given to model instances seems to be closely tied to the single representation to entity transformation. In the context of ongoing transformations between the domains the case for instances is much weaker. Process evolution also weakens the concept of a instance, where it refers to identifying the generic process description that an enactment conforms to. We argue that model instances are not fundamental to a PSS. It is just as valid to think of describing and enactment as enacting a description. Less emphasis on instances also encourages us to think more generally about process fragments, in both the representation and entity domains, and how they can be developed, composed, separated and re-composed. This is one of the motivations for our interest in incorporating the benefits of architecture description languages (ADLs) in process modelling languages and support systems [Cha00, See01]

The final observation from this paper is that there are system as well as modelling language issues in the design of a PSS. In [War98] we mentioned instances as an issue in our design of a second generation process language. Discussions at EWSPT'98 and the separation of process and language requirements discussed in [Sut95] made us review this. On reflection, many of the issues around instances concern providing a process system supporting many models over time, rather than the expressive power of a process modelling language. Our preference is for a PSS that is a virtual machine [Mor00b], and therefore closely related to the language it executes. However the issues discussed in this paper are important to any PSS system, irrespective of its chosen implementation technology.

## References:

[Cha00] Chaudet, C., Greenwood, R.M., Oquendo, F. and Warboys, B.C., Architecture-driven software engineering: specifying, generating, and evolving component-based software systems. IEE Proceedings – Software, to appear

[Dow94] Dowson, M., and Fernstr_m B.C., Towards Requirements for Enactment Mechanisms, in Proceedings of the Third European Workshop on Software Process Technology, April 1994, LNCS 775. pp. 90-106

[Fei93] Feiler, P.H., and Humphrey, W.S., Software Process Development and Enactment: Concepts and Definitions, in Proc. of the 2nd International Conference on Software Process, Berlin, 1993. pp. 28-40

[Fin94] Finkelstein, A. et al, *Software Process Modelling and Technology*, Research Studies Press, 1944, ISBN: 0863801692.

[Gre96] Greenwood, R.M., Warboys, B.C., and Sa, J., Co-operating Evolving Components – a Formal Approach to Evolve Large Software Systems, in the Proceedings of the 18th International Conference on Software Engineering, Berlin, 1996.

[Gre00] Greenwood, M., Robertson, I. and Warboys, B., A Support Framework for Dynamic Organisations, In the Proceedings of the 7th European Workshop on Softwre Process Technologies, LNCS 1780, Springer-verlag, Kaprun, Austria, February, 2000.

[Mor95] Morrison, R., Connor, R.C.H., Cutts, Q.I., Dustan, V.S., Kirby, G.N.C., Exploiting Persistent Linkage in Software Engineering Environments. *Computer Journal*, 1995, Vol 38 No 1, pp 1-16.

[Mor00a] Morrison, R., Linguistic Reflection: Introduction and State of the Art In Atkinson, M.P., and Welland, R., (Eds.) *Fully Integrated Data Environments*, Esprit Basic Research Series Springer-Verlag, Berlin, 2000, pp 155-156.

[Mor00b] Morrison, R., Balasubramaniam, D., Greenwood, R.M., Kirby, G.N.C., Mayes, K., Munro, D.S. and Warboys, B.C. A Compliant Persistent Architecture. *Software Practice and Experience*, Special Issue on Persistent Object Systems, 2000, Vol 30 No 4, pp 363-386.

[Oul95] Ould, M.A., *Business Processes: Modelling and Analysis For Re-engineering and Improvement*, John Wiley & Sons, 1995.

[PML] ProcessWise Integrator, PML Reference Manual, ICL/PW/635/0, April 1996. PML pre-defined class definitions available at http://processweb.cs.man.ac.uk/doc/pml-ref/Index.htm (accessed on 18 Jan 2001)

[PWeb] Process*Web* service and documentation http://processweb.cs.man.ac.uk/ (accessed on 18 Jan 2001)

[Sa95] Sa, J., and Warboys B.C., 1995. A Reflexive Formal Software Process Model, in Proceedings of the Fourth European Workshop on Software Process Technology, April 1995, LNCS 913.

[See01] Seet, W. and Warboys, B.C., A formal environment for specifying, generating and evolving enactable architecture based process support systems, submitted to EWSPT'01

[Sut95] Sutton, Jr., S.M., Tarr, P.L., and Osterweil, L. An Analysis of Process Languages, CMPSCI Technical Report 95-78, University of Maccachusetts, 1995

[War98] Warboys, B.C., Balasubramaniam, D., Greenwood, R.M., Kirby, G.N.C., Mayes, K., Morrison, R., and Munro, D.S., Instances and Connectors: Issues for a Second

Generation Process Language, in Proceedings of the Sixth European Workshop on Software Process Technology, September1998, LNCS 1487.

[War99a] Warboys B.C., Kawalek P., Robertson T., and Greenwood R.M., 1999. *Business Information Systems: a Process Approach*, McGraw-Hill, Information Systems Series, 1999, ISBN 0-07-709464-6.

[War99b] Warboys, B. (Ed.) Meta-Process. in Derniame, J.-C., Kaba, B.A., and Wastell, D. (Eds.) Software Process: Principles, Methodology, and Technology, 1999, LNCS 1500, pp.53-93

[Zir00a] Zirintis, E., Kirby, G.N.C., and Morrison, R., Hyper-Code Revisited: Unifying Program Source, Executable and Data, in Proc. 9th International Workshop on Persistent Object Systems, Lillehammer, Norway, 2000.

[Zir00b] Zirintsis, E., Towards Simplification of the Software Development Process: The Hyper-Code Abstraction, Ph.D. Thesis, University of St Andrews, 2000.