Cooperating Evolving Components
a rigorous approach to evolving large software systems*

R.M. Greenwood, B.C. Warboys
Department of Computer Science,
The University of Manchester, Manchester M13 9PL UK
email: markg,brian@cs.man.ac.uk
J. Sa
Department of Computing,
University of the West of England, Bristol BS16 1QY UK
email: j-sa@csm.uwe.ac.uk

Abstract

Large software systems have a large number of components and are developed over a long time period frequently by a large number of people. We describe a framework approach to evolving such systems based on an integration of product and process modelling. The evolving system is represented as a Product Tower, a hierarchy of components which provides views of the product at multiple levels of refinement. The evolution process is component based with the cooperation between components being mediated by the Product Tower. This ensures that the evolution process is scaleable and that it maintains, and evolves, the design model. We illustrate our approach with an example, outlining an evolution both of the product and of the process. The reflexive facilities of the process are shown to be key in ensuring the framework's ability to evolve.

Keywords: product evolution, process evolution, process modelling, design hierarchy

1 Introduction

Modern large software systems are systems which:

- Have a large number of components some of which are commodity, some are specially commissioned, and some are developed in-house.
- Are developed over a long time period, in fact Lehman's [13] classification of software system types argues that they never stop developing. His so-called *E-type* systems.
- Are developed by a large team of people.

Therefore any development support environment selected for managing the evolution of such systems needs to recognise that:

- For Method support:

  It must allow for the use of differing methods as some components will need to be developed by the use of a specific method

  It must allow for the methods used to evolve over time.

- For Tools support:

  It must allow for the introduction of new tools to aid the development

  It must allow for the versions of software tools used to develop components to vary.

- For System support:

  It must allow for the development of partial systems

  It must allow for the composition of all the currently defined components in order to check that the system, to date, is consistent. This includes both the functional specification of the components being built and the different methods used for their development.

Historically the development of large software systems has tended to separate the representation of the product being developed, typically some refinement hierarchy, from the process (and plan) used to manage that development. This has tended to mean that a monolithic approach to the management of development has been the norm. In order to keep control of a large development, a single management approach has been taken to all components.

Additionally the refinement hierarchy used to represent the design model has *only* been utilised as a design model. The fact that the design hierarchy represents an implicit model of the process used for development has not been properly exploited by the development process. With the development of Software Process Modelling techniques and tools, it is now possible to review this approach and see whether advantage can be gained by the exploitation of a more integrated method.

We illustrate our approach using OBM, a formalism based on temporal logic, to describe an Electronic Point Of Sale (EPOS) system and its component evolution process. This is implemented using ICL's ProcessWise Integrator, and an evolution both of the product and the process is outlined. In section 2 we describe reflexive OBM which is used to specify both the product and its associated evolution process. The *Product Tower*, the model of the design hierarchy, is illustrated in section 3 where we outline the OBM for the *application* part of the example EPOS system. In section 4 we again use OBM to specify hdev, which is the *method* part of each component in the tower. The way that the component hdev processes cooperate to control the overall evolution of EPOS is illustrated in section 5. In addition, the ability of hdev to evolve itself is illustrated. Finally we compare our approach with related work, and describe our conclusions and plans for future work.

2 OBM — the specification language

OBM [11, 19] is a formal development method. It provides a specification language. In the rest of this paper, we refer to both the language and the method as OBM. OBM is used to specify both the development process and the product to be developed. The advantages of using OBM include

- allowing different processes to be employed for developing different parts (and levels) of the product since OBM only defines the constraints on the ordering of activities
- allowing the process to evolve over time for any part of the product
- the ability to verify the consistency between levels in the product hierarchy since it is a compositional approach

2.1 Background of OBM

Originally, OBM was developed to specify concurrent systems which may have interference behaviour. The particular temporal logic we chose offered a compositional approach [3]. Using that approach, a system is considered as a collection of components. Each component is specified as a temporal logic formula. The specification of the overall system is obtained by composing all the component specifications. We found that the specification notation was rich enough to express the properties of the types of concurrent system with which we were concerned. However the specifications were not readily comprehensible. As a result, we gradually added more high level constructs to denote the temporal logic formulae. The collection of constructs was eventually evolved to a high level language with its semantics defined in the temporal logic.

In OBM, a system is considered to be composed of components which can be executed concurrently. The model of a system, in terms of the processes which it supports, is developed at several levels. At each level, the model contains a collection of component specifications.

OBM provides *abstract objects* to model processes in a system at a level of abstraction, and *boxed objects* to compose a number of components and to hide the internal operations of this composition.

An *abstract object* models part of a system which may be refined later. The behaviour of an abstract object is specified in terms of *operations*. An operation may be refined into several operations which may themselves be further refined. An abstract object *provides* operations and may *require* operations provided by other components. The *interface* of an abstract object consists of a list of provided operations, required operations and an operation pattern defined in terms of the provided operations. Operation patterns are defined using an *ordering expression*. The detailed definition of ordering expressions can be found in [18]. The *body* of an abstract object consists of a *call template* for each required operation and a definition for each provided operation which contains its type, i.e. active or passive, its *call pattern*, and its *pre* and *post-conditions*. Each call template includes: a name, an indication of whether or not the caller receives a result, the name of the component providing the required operation, and parameter details. Call patterns are defined using ordering expressions, and specify the constraints on the ordering of the calls made to the required operations. Call patterns are defined in terms of call template names. Pre and post-conditions are defined in terms of the parameter, the result and the values received from the calls made to the required operations.

The behaviour of an abstract object is determined by the execution of its operations. The order of these executions must conform to the operation pattern. The temporal semantics of abstract objects are given in [11].

An abstract object may be *refined* by decomposing it into a number of sub-components. In order to justify the refinement, it is necessary to show that the composition of the sub-components is consistent with the abstract object.

The level i+1 refinement of a level i abstract object consists of:

1. *Component Refinement:* replacing the level i abstract object by a number of level i+1 components, and defining each level i+1 component.

2. *Operation Refinement*: replacing the sub-patterns of the level i operation pattern by operation patterns defined in terms of level i+1 operations. A sub-operation pattern may also be a single operation. Given a level i+1 refinement, a level i+1 operation is *internal* if it is not used in the *Operation Refinement* part.

A *boxed object* is used for checking the consistency of refinements. It composes level i+1 components together to show that the composition is consistent with the level i abstract object. A boxed object contains a number of components and a list of internal operations. The behaviour of a boxed object is the concurrent executions of all the contained components. The behaviour of the internal operations is not visible. The temporal semantics of boxed objects, and how they are used in checking refinement consistency, are given in [11].

A *Product Tower* of components can be formed from a single top node by repeated refinement by decomposition. The ability to check the consistency of refinements is used to support the cooperative evolution of components.

2.2 Reflexive OBM

Based on the original OBM, a new feature is added so that the OBM development process is reflexive, that is it has the ability to modify itself, the notion was initially explored in [19].

Each object has two parts: the method part (or the meta-part) and the application part.

The method part defines how to develop the object. For example, the method part may describe that the first operation is to specify the application part, and after this, the application can be modified or refined. The method part may also include a decompose operation which creates new objects. Each new object will inherit its method part from its creator. An advantage of reflexive OBM is the ability to describe how the method part itself can change.

The application part is the result of applying the operations in the method part. The advantage of using OBM both as the specification language for the process and the product, to be produced by this process, is that we can apply a compositional approach to verifying the system at any level of granularity. Thus we can apply the development process (or some part of it) to even single operations in the product specification.

3 EPOS - an example Product Tower

We consider the development of a large system as being represented by a design hierarchy of components. We refer to this as a *Product Tower.* This tower we use as a complete representation of the development of a product. From it are referenced:

- Its complete design and implementation files. The *Application* part.

- The process which developed and continues to develop (*evolves*) each of its components. The *Method* part. This process also defines the toolset used and to be used to develop each component. This obviously varies with component type. A specification component might use a VDM process and an implementation component a C programming environment. The component type is implicitly defined by its associated process program.

At the most abstract level EPOS is simply one system (or component). This top level view can be summarised by the operation pattern

$$(trading^* \mid\mid managing^*); \circ$$

This says that EPOS can be viewed as an abstract object with two operations *trading* and *managing* which can both be repeatedly executed in parallel. During EPOS' development this top level is refined into three cooperating systems: TA, the Trading Application, TS, the Trading Support, and BO, the Back Office facilities. We refer to TA, TS and BO as the children on EPOS.

The OBM representation of the application part of TA is:

Abstract Object *TA* The Interface Part:

provides

$login, servecust, logout$

requires

TS:$logusers, payment$
BO:$deleteitem$

operation pattern

$$(login; servecust^{*}; logout)^{*}; \circ$$

This refinement of EPOS is verified by an operation refinement where

$$trading \longleftrightarrow login; servecust^{*}; logout$$

and *managing* is refined to a pattern consistent with the operations provided by TS and BO.

The symbols used in the above ordering expressions have the following meaning:

$P; Q$: P followed by Q.

$P \oplus Q$: either P or Q.

P^{*}: Zero or more occurrences of P.

$\circ$: end.

4 Hdev - an example evolution process

In order to achieve a scaleable approach we require a process which is based around evolving a component rather than a process which is specific to the structure of EPOS. The parent-child links in the *Product Tower* indicate points where there must be cooperation between the evolving components. In addition, we cannot assume that the tower itself will be immune from evolution; changing the tower structure is a natural part of EPOS's evolution.

Associated with each component is its childrens' *sub-towers*. Each component in a *sub-tower* is formed by enacting the process (method part) associated with its parent. A default process is inherited from its parent. Thus, at any point in time, the *Product Tower* represents both the state of the product in terms of components and their representations and the allowable process(es) which can legally be performed on each component. The *Tower* represents a refinement for every component which has children. Each refinement will be in a verified or non-verified state. These verification states are recorded with attendant implications for the parent and children involved. This means that each component in the *Tower* participates in at most two refinements: one with its parent, and one with its children. A component may be verified with respect to its children even though they are not verified with respect to theirs.

The default process, which we call hdev, consists of the following operations:

- *specify* - this changes a leaf component, that is one which has no sub-towers attached to it. After this operation the component will be non-verified with respect to its parent.

- *modify* - similar to *specify*, but used for components with sub-towers attached. After this operation the component will be non-verified both with respect to its parent and with respect to its children.

- *decompose* - decomposes a component. This creates a number of components, which become this component's sub-towers. All of these will be given the default process. If the component is a leaf then the *Product Tower* will grow by addition of the new components. If the component is not a leaf then its current children will be pruned and replaced with the new components. The created components inherit their method part from their parent.

- *verifydown* - this operation is used by a non-leaf component to verify its consistency with its children. It calls the *verifyup* operation of each of its children. If one or more of the children are currently amid another operation, then the *verifydown* will idle waiting until it can complete its *verifyups*. The result of a *verifydown* is either true, the object and its children are consistent, or false, they are inconsistent. A false result does not indicate the component which needs changing. It might be that a change to one of the children had simply been forgotten, or the verification might reveal a problem which needs addressing further up the tower.

- *verifyup* - this operation is called by *verifydown* of the component's parent.

- *changeself* - this operation enables the component to change its process. That is it allows the component to replace hdev with an alternative process. Whilst all the other operations manipulate the application part of components, *changeself* manipulates the method part.

In hdev, the operation pattern for the method part of a component is:

$$((doing \oplus verifying)^{*}; changeself; \circ$$

where

$$doing := (specify \oplus modify \oplus decompose)$$
$$verifying := (verifyup \oplus verifydown)$$

A component can either be actively developed (*doing*) or verified (*verifying*). These are exculsive options ($\oplus$) and are allowed repeatedly (*) until there is a decision to change the process (*changeself*). The above pattern does not explicitly identify all the limits on the orderings of the operations: there are additional preconditions on *modify* and *verifydown* to ensure that they are only done by non-leaf components, and on *specify* to ensure that it is only done by leaf ones.

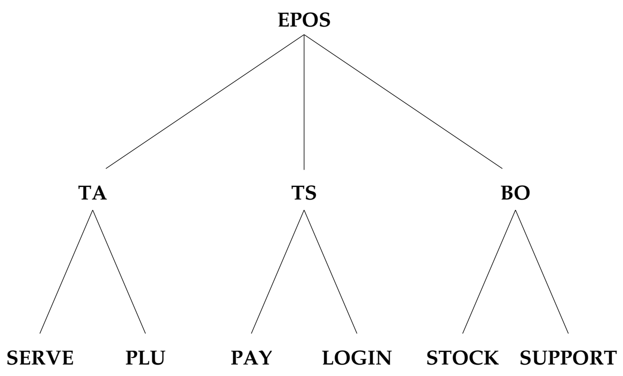

Figure 1: **Design Hierarchy for EPOS Product**

In our example, we will join the evolution of EPOS after the Product Tower illustrated in figure 1 has been established.

5 Applying hdev to EPOS

We have implemented the EPOS Product Tower with hdev as each component's method part as a process centred evolution environment implemented using ICL's ProcessWise Integrator (PWI) (which is briefly described below). The process hdev is implemented in PWI's process modelling language (PML). This forms an active model ensuring that the OBM for the EPOS application part is evolved as prescribed by the hdev process. In addition, the dynamic change capabilities of PWI's PML are used to implement *changeself* ensuring that hdev itself can evolve.

5.1 ProcessWise Integrator

ProcessWise Integrator (PWI) [4, 22, 25] originated in the Alvey IPSE 2.5 project which concluded in 1989. ICL subsequently took up the software rights and continued its commercial exploitation to the present day.

PWI's PML is based on a combination of Holt's Role Activity Diagrams [9] and Greenspan's RML [8]. The language has four principal classes: *entities* (records), *activities* (procedures), *roles* (encapsulations of data and behaviour), and *interactions* (unidirectional communication channels). These principal classes, along with numerous primitive classes, are used to define the process as a network of role instances. One important set of primitive classes is the *User Actions*; these enable process participants (users) to interact with the model.

A PWI environment has three elements: the Process Control Manager (PCM), the User Interface server, and the Applications servers. The PCM identifies, locates and schedules actions which can be executed. It does this by scrutinising each role, and examining the guard for each action to determine if it is currently ready, i.e. allowed to be executed. If there are *User Actions* ready then the PCM communicates with a User Interface server to present the user with an agenda of ready actions. The user chooses the next action from this list. Similarly if an action involving external tools is ready then the PCM communicates via the Applications server.

The support for tools integration, via the Applications server, is vital to our approach. The process model is not only supporting the orderings in the development process, and thus for example enforcing verification at each stage, but also providing the vital link between a component and the right version of the right tool to develop any component. This allows for rapid tool evolution in that differing components can utilise differing toolsets and wholesale backwards compatibility issues can be avoided. This is a key requirement in the development of large systems over long periods of time.

Process participants communicate with roles through a "pseudo-role" know as their *user agent*. The user interface is hierarchic: at the top level there is an agenda of roles bound to a participant's *user agent*, each of these opens into an action agenda.

5.2 Making a change to EPOS

In our example we will unrealistically have only one process participant, peter. This allows us to concentrate on the *Product Tower* aspects of the process rather than the multi-user ones. We will have a role for each EPOS component. Figure 2 shows the ten roles in peter's role agenda. (Each agenda entry has a symbol giving status information and the role name, for example SERVE is awaiting peter's attention (!!!)). The entry for TA has been opened to reveal its action agenda.

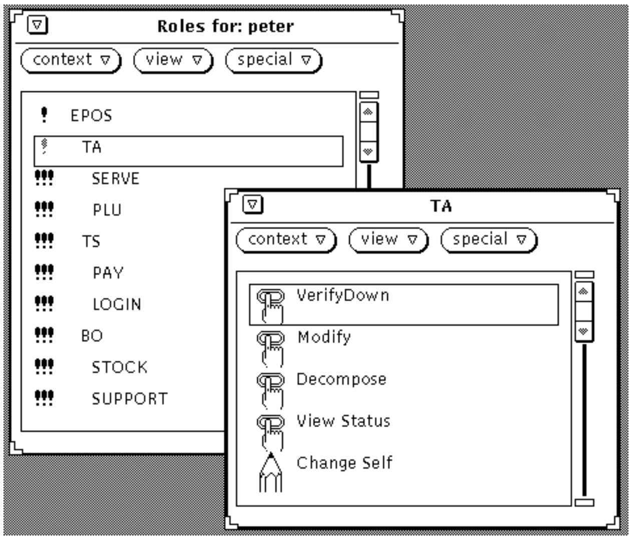

Figure 2: **PWI Role and Actions agendas**

This shows the actions available to a non-leaf component: *modify*, *decompose*, *verifydown*, and *changeself*. In addition, there is an additional action, *viewstatus*, which simply allows the participant to browse the current state of a component. (Each action appears in the agenda as an icon followed by a name.) An example output from *viewstatus* is given in figure 3.

The change to EPOS which we are going to consider is a result of a business decision. A supermarket chain using the EPOS software decides to offer its customers a store card to promote customer loyalty. Customers gain points in proportion to how much they spend, and bonus points when the card total first reaches certain amounts. These points are redeemed as money off future purchases. The chain is anxious for this change to be incorporated into their software as quickly as possible.

This change obviously concerns serving customers so TA appears to be an appropriate place to start. The operation pattern for TA is:

$$(login; servecust^*; logout)^*; \circ$$

TA is refined into SERVE and PLU. We must decide if we expect this structure still to hold, or if a new

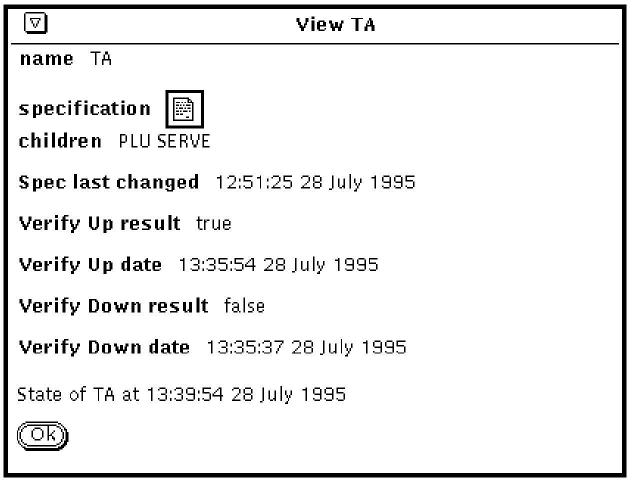

Figure 3: **view status output**

decomposition of TA is required. In this case we will assume that the points total is stored on a smart store card so the EPOS system needs a new operation to update the card, but not a new component to store the current points totals on cards. This means TA's operation pattern is not affected so we look at the operation refinement of *servecust*:

$$servecust \longleftrightarrow item^*; payment$$

Both *item* and *payment* are operations provided by SERVE, so we move down and focus on this component.

The current operation pattern for SERVE is:

$$(signon; (item^*; payment)^*; signoff)^*; \circ$$

As SERVE is a leaf component we will use *specify* to edit the OBM for its application part adding a new provided operation *bonuscard*. On selecting *specify* PWI will present the user with the OBM for SERVE in an editor window (alternatively it could call an external OBM editing tool). While editing the OBM for SERVE the user might return to TA and select *verifydown*. The advantage of this is that the *verifydown* will start, lock the other components (providing they themselves are not being altered), and then wait until SERVE's *specify* is completed. Figure 4 illustrates this. The user is still editing the OBM; the *specify* action in SERVE. The "Awaiting" action in TA's action agenda is a null-action which merely provides convenient feedback to the user on the progress of the *verifydown*.

One of the key features of PWI is the persistence of processes. The user could now logout of PWI, go home

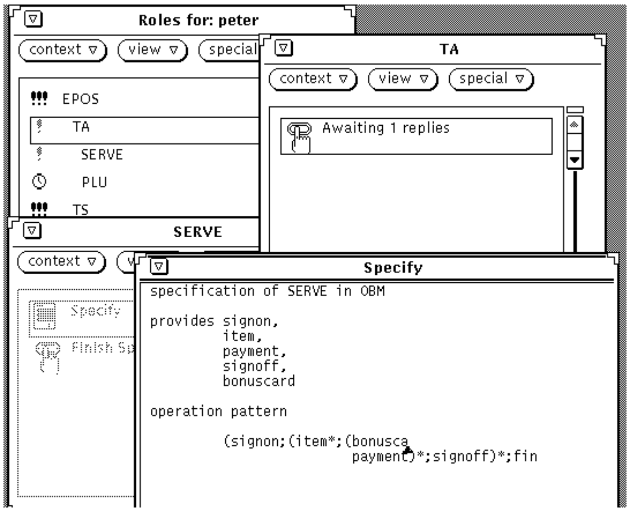

Figure 4: **SERVE specifying - TA waiting**

and return the next day to the process in the same state. Eventually the user will confirm that the editing of SERVE is complete, including its new operation pattern:

$$(signon; (item^*; (bonuscard; payment \oplus payment))^*; signoff)^*; \circ$$

SERVE will now be in a non-verified state with respect to its parent TA. As SERVE's *verifyup* is now available the pending *verifydown* will complete. Its result will either be false, or true given an appropriate change to the operation refinement of *servecust*. This consistency check will form an OBM boxed object from the new specification of SERVE and the specification of PLU, and test this is consistent with an operation refinement of TA. If the result is true then the refinement with parent TA and children SERVE and PLU is verified. If SERVE had children it would have to do a successful *verifydown* before being verified with respect to them.

The operation pattern of hdev indicates that verification is not enforced after every change to the product part of a component. This allows us to do several changes to a sub-tower before attempting to re-verify its consistency. An effect of this liberal approach is that children, being in a non-verified state with respect to their parent, covers both the case when there is a known inconsistency (an attempted verification proof failed), and the case when consistency is unknown because at least one component has changed.

While this change has been small it illustrates how our PWI implementation of hdev supports the evolution of EPOS through the interplay of component application changes and verifying refinements.

5.3 Changing hdev

We expect the EPOS components to evolve during the system's lifetime. In addition, we expect the set of components in EPOS' *Product Tower* to evolve. It is therefore only natural that hdev should evolve.

The key to hdev's evolution is the operation *changeself*. The effect of this operation is to change the process part of the component, including the operation pattern in which *changeself* appears. The *changeself* appears as the last operation in the pattern because after *changeself* the component will be at the start of the new operation pattern.

A simple example of *changeself* would be a change of policy so that verification was always done after any change; that is changing from the hdev pattern to:

$$\begin{aligned}((specify; verifyup) \oplus \\ (modify; verifydown; verifyup) \oplus \\ (decompose; verifydown) \oplus \\ verifydown \oplus verifyup)^*; changeself; \circ\end{aligned}$$

In the introduction we mentioned that a large software system would include different types of component. In our EPOS example so far we have only dealt with one type: OBM specification components. The specific set of component types will be influenced by the development methods adopted, and could potentially evolve during the system's lifetime. To illustrate this we will consider the scenario where the company evolving EPOS decides not to write their own stock management component, but to buy one in and integrate it with EPOS. The current OBM for STOCK provides the requirements which the acquired package must satisfy to form an effective part of the system.

This can be introduced into the EPOS Product Tower by:

1. decompose STOCK into a single component which we will call WHAREWIZ,
2. use *changeself* to replace WHAREWIZ's operation pattern. Two new operations are introduced:
   - *acquire* – purchasing the component and the rights to include it in EPOS,
   - *incorporate* – writing the interface routines between the rest of EPOS and the bought in component.

   and the new operation pattern is:

   $$acquire; (incorporate \oplus verifyup)^*; changeself; \circ$$

The principle of verification between STOCK and WHAREWIZ is the same as that between TA and its children, but the details will differ. It will probably be more of a review than a formal proof of consistency.

6 Related Work

In [13, 14] Lehman argues that the ongoing evolution of large (*E-type*) systems is inevitable. Similarly Parnas cites the problems of *Software Aging* [16], especially for successful systems. This is certainly the experience of VME/B [24] and supports the view that initial development is just one stage of product evolution.

The use of hierarchy to manage complexity is a central tenet of systems theory [21, 26] and has been incorporated in several software development methods, such as SADT [17] and CADES [25]. Experience with the latter also highlighted the inter-relationship between process and product in large systems evolution. The desire to support formal reasoning at multiple levels of abstraction has been one main aim in the development of OBM [18, 20, 11].

The *Product Tower* provides an explicit structural description of the software as a unifying framework. It is thus very similar to the approach advocated in [12], although in our case the full hierarchy may not be obvious from the final product.

Process modelling, and particularly the use of explicit process models in a software engineering environment, has grown as a research area since Osterweil's keynote speech at ICSE 9 [15]. A good introduction is provided in [6], and [7] describes a range of systems. In general the style is project-based. It is expected that a software development company would have a set of generic process models which are specialised to the task in hand. An effect of this is that attempts to address large systems have resulted in large and complex process models [1, 5]. This contrasts with the component-based approach descibed in this paper. We are able to keep the component process simple by re-using the information in the *Product Tower* to identify where cooperation is required. The use of a product representation which is aligned with a process model is also described in [10].

In a project-based approach, process evolution is often viewed as the feedback which is incorporated in the generic processes ready for future specialisations. Having a single process to support the ongoing evolution of a component means that the process must evolve itself; it must be reflexive. The ability to cope with such reflexive processes was a major design criteria of PWI, and examples exploiting this are described in [25, 23]. The SLANG process modelling language also adopts a similar approach to evolving processes [2].

7 Conclusions

Our prime objective was the production of a *framework* within which large software systems could be developed. We have shown that this *framework* is both implementable and verifiable.

An interesting effect of modelling both the product and the process in the same notation is the ability to expand from a process product pair into chains. For example we could use a bootstrap process to develop a software development environment as a product and then use that product as a process to develop a real end-user product. As software systems become more integrated with the businesses which they support so the demand for software products which can evolve in real-time with the business need will grow. This will lead to a requirement to include in software products the ability to evolve and a component with the properties of the type of framework which we describe will, we believe, become a necessity.

The important properties of this *framework* from the perspective of the software engineering of large software systems are that:

- By utilising the product design hierarchy as the reference point for all information about the development of a product, we can guarantee that the approach is scaleable. Provided that a hierarchy of components can be produced, we can produce a *matching* process development hierarchy.

- Approaching the issue from this perspective ensures that once a new component has been conceived, it will *inherit* the basic process from which control derives.

- The process which supports the verification of a design refinement allows for the presence of *non-validated* and hence partial components.

- The ability to specialise the process for any component allows components to be developed using differing methods and indeed different versions of development tools. For bought-in components the process merely records their acquisition process.

- Evolution of the developing system is managed through the effects of evolution on the design hierarchy. The verification process checks the *rippling* of change *downwards* through the hierarchy. That is the effects which any change has on the rest of the system. The verification can be activated from any node in the design hierarchy since the process for verification is inherited when components are decomposed. Thus small sub-trees can be validated in addition to the whole system.

8 Future work

In this paper we have established a *framework* mechanism for the support of large software systems evolution. The properties of large software systems which we have not directly addressed are support for:

- multiple users
- component versioning
- component acquisition
- consistent process evolution

The mechanism for attaching to a component, the specific process for developing that component, also provides us with the basis for addressing these issues.

Multiple users are supported by the mechanisms provided in the ProcessWise Integrator. Each process fragment, the *method part*, attached to each component, can be assigned as a role to different and multiple users. The process would naturally be extended to deal with co-operating issues such as locking.

Component versioning is supported by the creation of multiple instances of each *sub-tower*. The *method part* associated with the owning component retains the knowledge of these multiple instances and further, since it *is* the process which created the new instance(s) it also has knowledge of the reasons for creating a new version. Thus multiple versions can be retained and selected as *sub-towers* by their purpose rather than by just the conventional version number. On encountering a multiple version *sub-tower* the owning *method part* will provide both *doing* and *verifying* processes with the available options.

Component acquisition will be supported by providing the ability to add and move *sub-trees* around the Product Tower. Some of this is readily achievable, and indeed was illustrated by the acquisition of WHAREWIZ for EPOS, in section 5.3 above.

Consistent process evolution is a complex task and we will continue to develop the mechanisms for managing the evolution of the development process. Development processes are, after all, subject to the same kind of evolution as products. Our work to date, in this area, has concentrated on the specification of a *meta-process* for managing these changes. The work is based on the notion of a Process Model for Management Support (PMMS), which originated during the IPSE 2.5 project [25], and which was further developed in [23]. The *framework* reported here gives us a general basis for managing the way in which an appropriate development process is attached to a new component. In the current system this process is *inherited* from its parent component and can be changed using the *changeself* construct. This clearly leads to the possibility of inconsistent processes since *changeself* can clearly change the process into anything. Our work on *meta-process* is aimed at providing a clean management of process evolution to complement the *framework* for managing product evolution. The *changeself* mechanism being replaced by a PMMS style management model.

Acknowledgements

This work was partly supported by UK EPSRC projects GR/J11034 and GR/J49013. Thanks are also due to our colleagues at the Informatics Process Group, Manchester University, at Southampton University, and in the ESPRIT working group PROMOTER.

ProcessWise Integrator is a trademark of International Computers Limited.

References

[1] J.-M. Aumaitre, M. Dowson, and D.-R. Harjani. Lessons Learned from Formalizing and Implementing a Large Process Model. In *Proceedings EWSPT'94*, volume 772 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[2] S. Bandinelli and A. Fuggetta. Computational Reflection in Software Process Modelling: the SLANG Approach. In *Proceedings of ICSE 15*, Baltimore, Maryland, 1993.

[3] H. Barringer, R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. In *Proceedings of the 16th A.C.M. Symposium on Theory of Computing*, 1984.

[4] R.F. Bruynooghe, R.M. Greenwood, I. Robertson, J. Sa, R.A. Snowdon, and B.C. Warboys. Padm: Towards a total process modelling system. In A. Finklestein, J. Kramer, and B. Nuseibeh,

editors, *Software Process Modelling and Technology,* pages 293–334. Research Studies Press, 1994.

[5] D.C. Carr, A. Dandekar and D.E. Perry. Experiments in Process Interface Descriptions, Visualization and Analyses. In *Proceedings EWSPT'95*, volume 913 of *Lecture Notes in Computer Science*, pages 119–137, Springer Verlag, 1995.

[6] B. Curtis, M.I. Kellner and J.Over. Process Modeling. In *Communications of the ACM*, 35 (9): 75–90, 1992.

[7] A. Finklestein, J. Kramer and B. Nuseibeh, editors. *Software Process Modelling and Technology.* Research Studies Press, 1994.

[8] S.J. Greenspan, Requirements Modelling. In *Technical Report CSRG-155, University of Toronto, March, 1984.*

[9] A.W. Holt, H.R. Ramsey, and J.D. Grimes. Coordination system technology as the basis for a programming environment. In *Electrical Communication*, 57(4):308–314, 1983.

[10] K.E. Huff. On the Relationship between Software Processes and Software Products. In *Proceedings of the 6th International Software Process Workshop*, pp. 109–111, Hakodate, Japan, 1990.

[11] J.A.Keane, J.Sa and B.C.Warboys, Applying a Concurrent Formal Framework to Process Modelling. In *Proceedings of FME'94*, volume 873 of *Lecture Notes in Computer Science*, pages 291-305, Springer Verlag, 1994

[12] J. Kramer Exoskeletal Software - Panel Presentation In *Proceedings of ICSE 16*, Sorrento, Italy, 1994

[13] M.M.Lehman, Programs, life cycles, and laws of software evolution. In *Proceedings of the IEEE 68, 9 (1980) 1060–1076*

[14] M.M. Lehman. Software engineering, the software process and their support. In *Software Engineering Journal*, 6 (5): 243–258.

[15] L.J. Osterweil. Software Processes are Software Too. In *Proceeding of ICSE9*, Monterey, California, 1987.

[16] D.L. Parnas. Software Aging. In *Proceedings of ICSE 16*, Sorrento, Italy, 1994.

[17] D.T. Ross and K.E. Schoman Jr. "Structured Analysis for requirements definition". In *IEEE Transactions on Software Engineering*, 3 (1), 1977.

[18] J. Sa and B.C. Warboys. Specifying Concurrent Object-based Systems using Combined Specification Notations. In *Technical Report UMCS-91-9-2, Department of Computer Science, University of Manchester*, July 1991.

[19] J. Sa and B.C.Warboys. A Reflexive Formal Software Process Model. In *Proceedings EWSPT'95*, volume 913 of *Lecture Notes in Computer Science*, pages 241–254. Springer Verlag, 1995.

[20] J. Sa and B.C. Warboys. Modelling Processes Using a Stepwise Refinement Technique. In *Proceedings EWSPT'94*, volume 772 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[21] H.A. Simon. *The Sciences of the Artificial.* MIT Press, second edition, 1981.

[22] R.A. Snowdon. An Introduction to the IPSE 2.5 Project. In *ICL Technical Journal* 6 (3): 467–478, 1989.

[23] R.A. Snowdon. An Example of Process Change. In *Proceedings EWSPT'92*, volume 635 of *Lecture Notes in Computer Science*, pages 179–195. Springer Verlag, 1992.

[24] B.C. Warboys. VME/B a model for the realisation of a total system concept. In *ICL Technical Journal* November 1980.

[25] B.C. Warboys. The IPSE2.5 Project: Process Modelling as the Basis for a Support Environment. In *Proceedings of the First International Conference on System Development Environment and Factories*, Berlin, May 1989

[26] B. Wilson *Systems: concepts, methodologies, and applications.* John Wiley and Sons Ltd, 1984.