A Reflexive Formal Software Process Model

J. Sa
Department of Computing,
University of the West of England, Bristol
email: j-sa@csm.uwe.ac.uk

B.C. Warboys
Department of Computer Science,
The University of Manchester, Manchester
email: brian@cs.man.ac.uk

Abstract. In this paper a very simple reflexive formal software development method is described. The method is called OBM which provides a formal specification language. This paper demonstrates that the OBM development method can be defined as a process model in the OBM language. An example is used to illustrate how to develop and modify applications using the OBM development method.

1 Motivation for the Study

The study of software processes has attracted increasing attention in the last decade. One of the characteristics of such processes is that they are subject to frequent change. Thus, often either a software process needs to be changed or a new process needs to be created. The activities for managing a process, e.g. creating or changing, are usually referred to as meta-activities. A process consisting of the meta-activities for a particular process is referred to as a *meta-process*.

As pointed out in [6], since meta-processes are also processes, they themselves also need to be created and controlled. What is this meta-meta-process for producing a meta-process? Potentially, there is an infinite chain of meta-processes. This problem can be avoided if a process provides activities with which to manage itself. This kind of process is called a *reflexive* process. A model of such a process is called a reflexive model.

There have been several studies of reflexive process models. The aim of our study is to define a reflexive process model which has a formal mathematical foundation. Such a formal foundation offers the opportunity to be able to reason about the properties of a process model before the model is enacted.

2 Objective of the Paper

In this paper a very simple reflexive formal software development method is described. The method is called OBM which provides a formal specification language. In the rest of this paper, we sometimes refer to both the method and the language as OBM. The objective of this paper is to show that the OBM development method can be defined as a process model in the OBM language. Further more, this paper will illustrate how to use this model to develop an application model (which can either be the specification for a software product or a conventional process model), modify it or change the development process itself.

3 Background of OBM

Originally, OBM was developed to specify concurrent systems which may have interference behaviour. Among the available formalisms that were suitable for describing such systems, we decided to use temporal logic because it was capable of expressing liveness properties [13]. 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 very comprehensible. As a result, we gradually added more and 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.

According to the definitions given in [18], OBM may be classified as an object-based language. However our motivation for defining OBM was not to provide yet another object-oriented language, but to make it easier to specify systems using temporal logic.

In OBM, a process is considered to be composed of components which can be executed concurrently. The model of a process 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. It *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 [14]. 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 is of the form:

$!name$	
prop	$func/proc$
obj	*another*
apar	v

where $!name$ is the name of the template, "prop" defines the type of the call, e.g. whether the caller receives a result, "obj" states the name of the component providing the required operation and "apar" specifies the actual parameter.

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.

The consistency checking of refinements is described in [11].

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 are defined in [11].

4 The Reflexive OBM

Based on the original OBM, a new feature is added so that the OBM development process is reflexive.

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 (meta-)operation is to specify the application part, after this, the application can be modified or refined, or the method part itself can be changed.

The application part is the result of applying the (meta-)operations in the method part. It defines the process to be modelled, for example, in this paper we use the example of a supermarket.

5 The Basic Reflexive OBM Development Process

Since the OBM development process is reflexive, the process itself can be changed. In this section, the basic (or the default) OBM process for developing an object (component) and its refinement is described.

To develop the application part of each component, the process is to:

- specify the component. Each component can be modelled as either an abstract object or a single object.
- verify that the specification is valid with respect to the specification in the level above.
- modify the specification of the component.
- refine the specification of the component. A refinement will create more OBM objects. Each object has two parts: the method part and the application part. The method part will be inherited from the parent when the object is created. Because this contains the ability to change self, the method can be modified and subsequently inherited by further refinements. The application part will be created using the method part.
- verify that a component's refinement is valid.
- change the OBM development process itself.

6 OBM definition of the OBM Development Process

Let's consider each OBM development process as an abstract object which provides several operations, for example, specify, modify, decompose and change self. The following is the definition of the interface of the application part. The operation pattern for the method is the same as the one for the application part.

<u>Abstract Object</u> $OBMdev$ <u>The Interface Part:</u>

provides

$$specify, verifyup, modify, decompose, verifydown, changeself$$

requires

$$none{:}none$$

operation pattern

$$specify; verifyup; (modify \oplus decompose; verifydown)^*; changeself; \circ$$

The above defines the interface part of the abstract object called $OBMdev$. It provides six operations: $specify$, $verifyup$, $modify$, $decompose$, $verifydown$ and $changeself$. It does not require any operations. The allowed execution order is specified by an ordering expression.

The symbols used in the above ordering expression 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.

The execution order of the above operation pattern is as follows: the first operation is $specify$; the specification must be verified against the level above; following the verification, there may be several iterations of one of two operations: $modify$ and $decompose$; a decomposition must be verified; before each iteration, the object $OBMdev$ may change itself.

7 An Example

In this example, we are going to explain how an application model can be developed and modified using OBM and how the basic OBM development process itself may be changed.

7.1 Specifying an Application Model using OBM

As mentioned in section 4, for each object in OBM, there are two parts: the method part and the application part. Let the operation pattern of the method part be the same as the one defined in section 6, i.e.

$$specify; verifyup; (modify \oplus decompose; verifydown)^*; changeself; \circ$$

Figure 1 illustrates that a system is to be developed. The enabled meta-operation is $specify$. (Enabled operations are underlined in the figures.) The application to be considered in this example is a supermarket system which deals with the back office management activities and the checkout activities. The supermarket system can be modelled as an abstract object which provides two operations: *management* and *trading*. These two operations can be repeatedly executed in parallel.

The supermarket model is developed or executed by executing the pattern of the method part. In this case, the first allowed (meta-)operation is $specify$. After doing $specify$ we will have a definition of the application part. Let the definitionls be as follows:

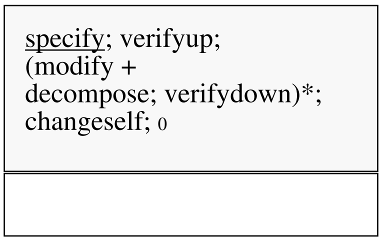

Fig. 1. Starting Developing a System

Abstract Object *supermarket* The Interface Part:

provides

$$management, trading$$

requires

$$none{:}none$$

operation pattern

$$(management^*||trading^*); \circ$$

7.2 Refining an Application Model

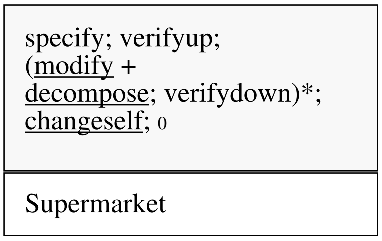

Fig. 2. After Specifying the Top Level

After specifying the top level, the allowed meta-operation is $verifyup$. Since this is the top level, the specification is by default valid. Figure 2 shows the situation after the application part has been specified and verified. To further develop the supermarket model, we continue to execute the method pattern. After $specify$ and $verifyup$, the allowed meta-operations are: $Modify$, $decompose$ or $changeself$. Assuming we choose to decompose the model.

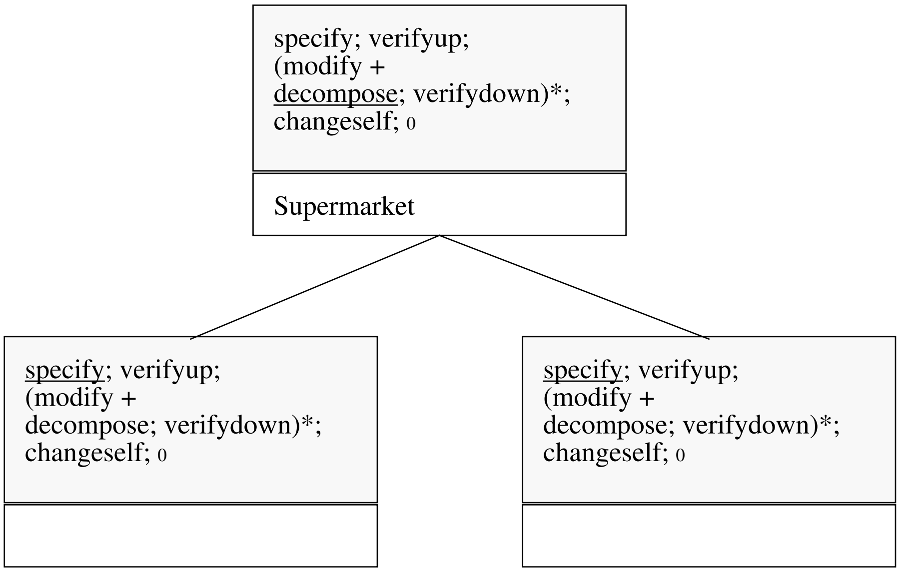

Fig. 3. Decomposing the Top Level

The operation $decompose$ will create new abstract objects. In this particular case, the supermarket model is decomposed into two components: back office and checkout. So, two abstract objects are created. When a new object is created, the method part is inherited. See figure 3.

Consider the checkout component, it is defined as an abstract object. The pattern of the method part is

$$specify; verifyup; (modify \oplus decompose; verifydown)^*; changeself; \circ$$

Similar to the supermarket model, the checkout model is developed by executing the pattern of its method part. Again, the first operation is to specify the checkout model. The following is a specification of the checkout component.

Abstract Object *checkout* The Interface Part:

provides

$$login, servcust, logout$$

requires

$$backoffice{:}someoperations$$

operation pattern

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

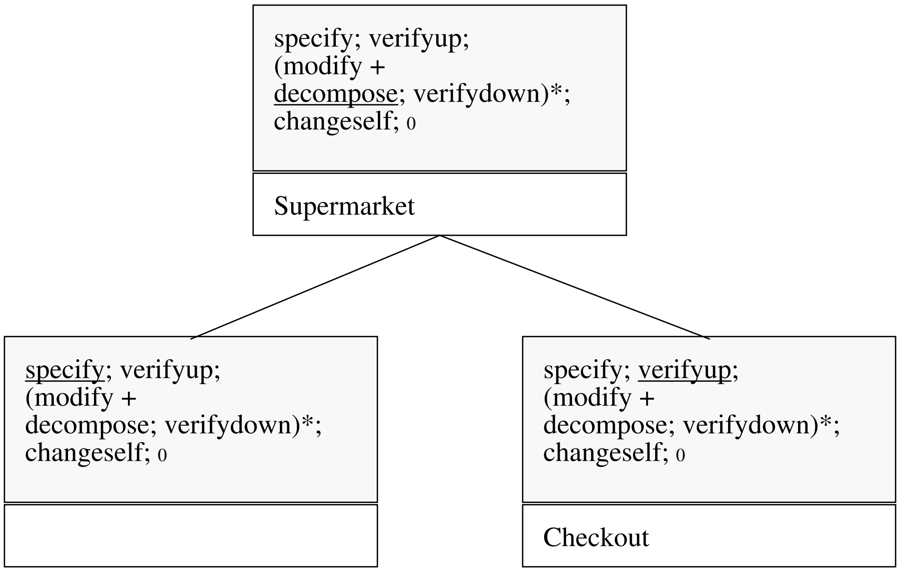

Fig. 4. After Specifying Checkout

The above specification defines that *checkout* provides three operations: *login*, *logout* and *servcust*. It requires some operations provided by the back office. The execution order specified by the operation pattern defines a repeated trading pattern. In each pattern, one is allowed to login, serve customers repeatedly, and then logout. See figure 4.

7.3 Verifying the Decomposition

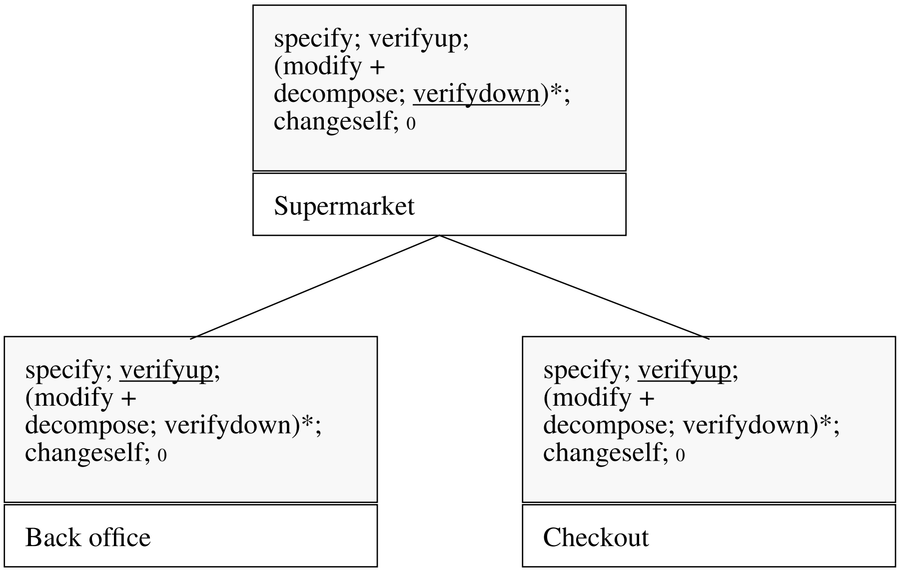

Fig. 5. Verifying the Decomposition

The decomposition must be checked for its consistency, i.e. we must prove that the composition of back office and check out is consistent with the supermarket node. This type of proof is described in [11]. Figure 5 shows that the supermarket node is performing the verification meta-operation. This operation calls the meta-operations $verifyup$ of both back office and checkout.

7.4 Modifying an Application Model

Consider the checkout node, after specifying and verifying the checkout component, we can continue to develop the checkout model by executing the pattern of the method part. For example, we can use $modify$ to change the operation pattern of the application part to

$$(login; (servcust; logtransaction)^*; logout)^*; \circ$$

With this new pattern, each service to a customer must be logged.

The modification of the application part of a node may trigger other changes:

- any node that uses this node, e.g. back office, may need to be changed
- all child nodes must be re-developed

After a modification, the consistency with respect to the level above, e.g. the supermarket node, must be checked again. The proof procedure is defined in [11].

7.5 Changing the Method Part

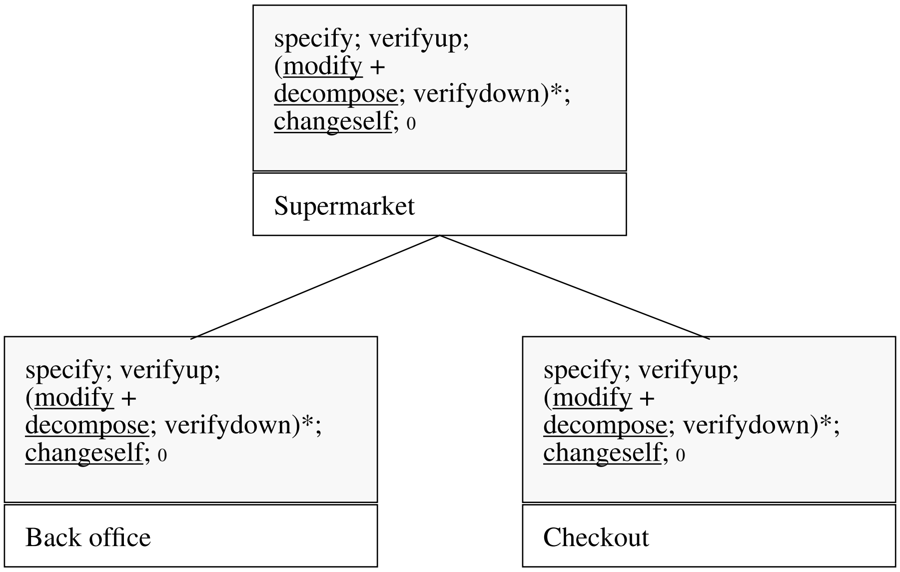

Fig. 6. After Modifying

After modifying the application, we again have the choice of doing either $modify$, $decompose$ or $changeself$. See figure 6. Let's choose $changeself$ to change the method part of the checkout node. For example, we may want to be able to execute the specification. So the new pattern for the method part may be

$$specify; verifyup; (modify \oplus execute \oplus decompose; verifydown)^*; changeself; \circ$$

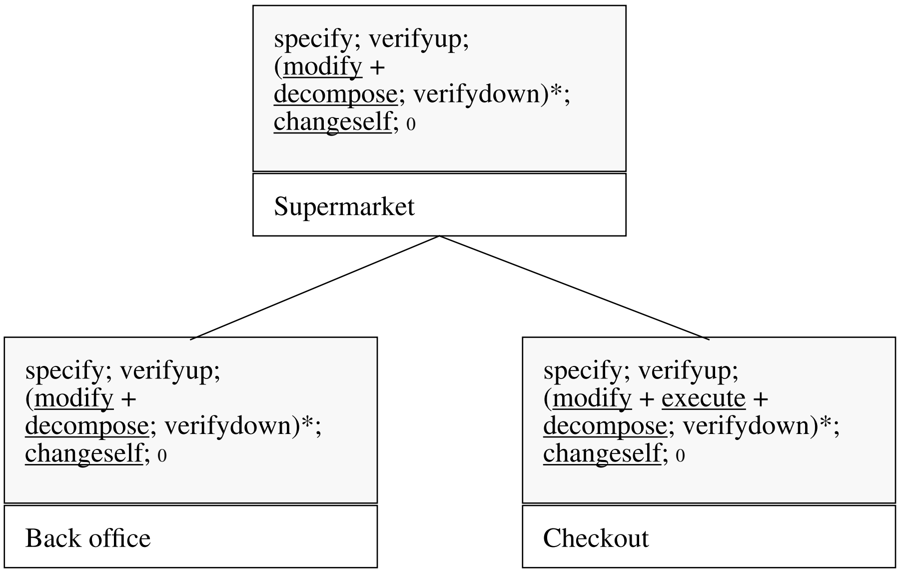

Fig. 7. New pattern for Checkout

In OBM, the meta-operation $changeself$ for each node must ensure that the application being developed will remain consistent. The consistency check involves:

- re-specify the application part of the node;
- re-specify the application part of all the child nodes;
- prove that the composition of the new child specifications is consistent with the new specification of the node (according to the procedures described in [11]). The method part can only be changed if the consistency is maintained.

After a *changeself*, all the child nodes will inherit the new pattern. The currently enabled meta-operation will be identified. The child nodes will continue their development by following the new pattern.

8 Related Work

8.1 Meta-Processes

PMMS PMMS (Process Model for Management Support) [17][15] was developed under the IPSE 2.5 project. It reflects a simple conceptualization of the process of establishing and managing activity. In response to some stimulus, i.e. the goal, it is decided to establish some activity. The method used to carry out such activity needs to be determined. This produces what is often called a plan. The plan is then installed and carried out. PMMS is very similar to our approach as described above. It is able to change the application model. It is able to change the process itself. There is a PMMS for each new goal, i.e. a new object. However PMMS has no formal foundation.

HFSP In [16], a meta-process model, called HFSP, is described. HFSP is a process model which describes software processes as a collection of activities which are characterised by their input and output relationship defined as mathematical functions. Meta-operations are introduced for handling the dynamism and flexibility of software processes. For example, when an execution of an activity is unsatisfactory, the meta-operations can be used to change the process dynamically.

This kind of change modifies what we referred to as the application part in the above example. A similar comparison is given in subsection 7.4. However it does not provide any mechanism for controlling the meta-process itself. For example, if the meta-process is not suitable for a particular project, what can be done about it?

EPOS In [9], a meta-process is defined which is capable of changing different types of models, including the meta-model schema itself. The framework described in [9] does not provide formal support for validation or verification of process evolutions. Although in this paper, we have not discussed about how to validate or verify process changes, since the OBM language has formal semantics, it is possible to perform formal reasoning in our approach.

8.2 Specification of Interference

In [10], a method for specifying concurrent behaviour with possible interference is defined in terms of `rely` and `guarantee` conditions. The method is used to specify a system for shared variables. In OBM, communication and interaction between different components can only be achieved via operation calls.

The only possible interference comes from receiving results from an operation call to another component. Therefore, to address interference, it is only necessary to consider results received from the operation calls. This means that by slightly modifying the well known and widely accepted notation - *pre-* and *post-*conditions, we are able to embed the specification of interference.

8.3 Multiple Levels of Abstractions

The need for modelling processes in multiple levels of abstraction has been clearly recognised. Dowson pointed out in [7] that modelling formalism should accommodate a wide range of model granularity, and allow the refinement of initially large-grain models to address increasing details. The example above has shown that OBM has achieved this requirement.

Several existing approaches also provide means to represent a model in multiple levels of abstraction. Some of these are based on hybrid formalisms that use different notations for large-grain and small-grain aspects of process. An example of this type of approach is Process Weaver [8]. Another type of approach is incremental definitions. Examples of this type of approach include: tasks in Epos [5], activities in Slang [2], blackboard in Oikos [1]. However there is no support for formal reasoning between different levels of abstraction. Since Slang is based on Petri-nets, it should be possible to perform formal reasoning, however, it is not clear to the authors whether such a facility is provided by [2]. Work on consistency proof for refinement in Oikos is in development [1, 4].

In [12], a method based on data flow diagram (DFD) for modelling process interface is defined. Although the method has a formal basis for process decomposition, it provides very limited expressive power.

9 Conclusion

This paper has illustrated two main points. Firstly, it is possible to change an application process model dynamically in OBM. Secondly, it is possible to change the method through the OBM method itself. Our work on this issue is on-going, there is still a wide range of improvement and future work required. However we believe we have demonstrated that these two powerful issues can be addressed in a fairly simple framework.

References

1. V. Ambriola and Carlo Montangero. Oikos at the Age of Three. In *Proceedings EWSPT'92*, volume 635 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.
2. S. Bandinelli, A. Fuggetta, and S. Grigolli. Process Modelling In-the-Large with SLANG. In *Proceedings of the Second International Conference on the Software Process*, Berlin, 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. X.J. Chen and C. Montangero. Compositional Refinement in Multiple Blackboard Systems. In *ESOP'92 European Symposium on Programming*, February.
5. R. Conradi et al. Design, Use and Implementation of SPELL, a Language for Software Process Modeling and Evolution. In *Proceedings EWSPT'92*, volume 635 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.
6. R. Conradi, C. Fernström and A. Fuggetta. A Conceptual Framework for Evolving Software Processes. in *Software Process Modelling and Technology*, A. Finkelstein, J. Kramer and B. Nuseibeh (Eds.), Research Studies Press, Wiley, 1994.
7. Mark Dowson. Software Process Themes and Issues. In *Proceedings of the Second International Conference on the Software Process*. IEEE Computer Society Press, 1993.
8. C. Fernström. PROCESS WEAVER: Adding Process Support to UNIX. In *Proceedings of the Second International Conference on the Software Process*, Berlin, 1993.
9. M.L.Jaccheri and R.Conradi. Techniques for Process Model Evolution in EPOS. in IEEE TSE.
10. C.B. Jones, Tentative Steps Toward a Development Method for Interfering Programs, *ACM TOPLAS* 5(4), 1983.
11. J.A.Keane, J.Sa and B.C.Warboys, Applying a Concurrent Formal Framework to Process Modelling. In *Proceedings of FME'94*, Barcelona, October, 1994. Pages 291-305, LNCS 873.
12. C. Kung. Process Interface Modelling and Consistency Checking. *Journal of System and Software*, 15:185-191, 1991.
13. L. Lamport. What Good is Temporal Logic? In R.E.A. Mason, editor, *Information Processing 83*, pages 657-668, IFIP, 1983.
14. J. Sa and B.C. Warboys. Specifying Concurrent Object-based Systems using Combined Specification Notations. Technical Report UMCS-91-9-2, Department of Computer Science, University of Manchester, July 1991.
15. R. Snowdon. An Example of Process Change. In *Proceedings EWSPT'92*, volume 635 of *Lecture Notes in Computer Science*, pages 179–195. Springer Verlag, 1992.
16. M.Suzuki and T.Katayama. Meta-Operations in the Process Model HFSP for the Dynamics and Flexibility of Software Processes. in Proceedings of the First International Conference on software Processes, 1991, pages 202–217.
17. B. 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
18. P. Wegner, Dimensions of Object-Based Language Design. in *OOPSLA'87 Proceedings*, October, 1987.