Process Modelling: a critical analysis

Anthony Finkelstein, Jeff Kramer & Matthew Hales
Imperial College, Department of Computing, 180 Queens Gate, London SW7 2BZ.

0 Abstract

This paper uses a software development environment construction case study as a framework for a critical analysis of software process modelling. It outlines a research agenda based on this analysis.

1 Introduction

Software process modelling (aka process programming) has assumed considerable importance in discussions of software engineering. In particular attention has been paid to the use of software process modelling in the construction of software development environments.

Despite the growing literature on this topic almost no independent critical analysis or evaluation has been available. This paper attempts to fill that gap. In particular we will be reflecting on experience with the Marvel environment from Columbia University. Marvel is the paradigm case of the software process modelling approach to building software development environments. In this paper we examine Marvel's strengths and limitations and look in detail at a small example of its use. We use this analysis as a basis for suggesting a research agenda for software process modelling.

2 What is Software Process Modelling?

Essentially, software process modelling is the construction of an abstract description of the activities by which software is developed. In the area of software development environments the focus is on models that are enactable, that is executable, interpretable or amenable to automated reasoning.

A particular "instance" of the software development process - the development of a particular piece of software - can be seen as the "enaction" of a process model. That model can be used to control tool invocation and interworking. A software development environment for a particular development is thus built up around (or generated from) an environment kernel which is essentially a vehicle for constructing and enacting such software process models. For a full discussion reference may be made to the extensive literature on this topic particularly to the Proceedings of the Software Process Workshop series (Potts 1984, Wileden & Dowson 1986, Dowson 1987, Tully 1989, Perry 1990, Dowson 1991).

The relation between software process modelling and reuse is complex. On the one hand models can be produced of software processes which incorporate reuse. An example of this will be discussed below. On the other hand software process models are themselves reused. A software process model should encapsulate valuable organisational knowledge about the conduct of the development process. An example of this is the reuse of complex optimisation strategies (Wile 1983). A further possibility is the construction of software process models in representation schemes that permit reuse, for example in object-oriented languages. These relations have been

relatively little examined. All rest on an accurate assessment of the benefits and problems of software process modelling as a whole.

3 What is Marvel?

Marvel is a software development environment kernel in the sense described above. An environment that has been produced with the Marvel kernel will be referred to as a Marvel environment. Marvel is the primary product of a research programme run by Gail Kaiser of Columbia University, New York. The goal of the programme is to "develop a kernel for generating multi-user development environments that use knowledge about the software development process of large-scale projects".

Marvel is based on experience with a multi-user programming environment called Smile (Kaiser & Feiler 1987). In Smile the description of the programming process was "hard-coded" into the environment. Marvel generalises this approach by providing support for the definition and enaction of software process models. Marvel environments run on top of the Unix family of operating systems.

There have been a number of versions of Marvel: version 0 was a "proof of concept" implementation which simply replaced the Smile process description by a separate "strategy" description; version 1 is the first large-scale implementation, based on the database manager developed for Smile; version 2 is a single-user implementation, independent of Smile, which includes a purpose built database manager; version 2.6, on which the example discussed below was based, includes some support for management of persistent data; Marvel 2.6 is supported on Sun-OS 4.0 with X-11 windows (also Ultrix 3.1 & AIX 2.2.1).

Marvel has been extensively described in the literature. Particular reference may be made to the key papers: Kaiser, Feiler & Popovich (1988) and Kaiser, Barghouti, Feiler & Schwanke (1988).

4 How does Marvel work?

To construct a Marvel environment the developer must produce a data model and a process model. The data model describes the objects to be managed during the process of software development and their properties. The process model describes the activities carried out on those objects by the developers and tools involved in the specified development.

Marvel uses the data model to generate an "objectbase" in which all artefacts (code, documentation, test cases and so on) created during a development are held. The objectbase also maintains history and status of the objects. The data model gives the types, or classes, of the objects involved, their attributes and the relationships between them. The objectbase is implemented straightforwardly as a Unix file structure. Each object instance has associated with it a unique directory, and directories are structured according to the relationships between the object instances.

The process model is given in the form of rules each of which gives the preconditions which must be satisfied if the activity is to be carried out; the activity; and the postconditions, in terms of the effects of the activity on the objectbase (conditions and effects in AI planning system terminology).

The Marvel kernel provides a means of enacting the process model. It does so in an "expert-system-like" manner by opportunistic processing. If the preconditions of an activity are satisfied that activity will be invoked, this may in turn result in the satisfaction of the preconditions of further activities and by *forward chaining* they will be invoked in turn. If a particular activity is chosen by a user and is not eligible for invocation the Marvel kernel will *backward chain* invoking those activities necessary for the selected activity to be performed.

Both data and process model are expressed as "strategies" in the Marvel Strategy Language. Strategies can be imported into a main strategy. It is standard to define the data model in a single strategy but have multiple process model strategies for related tool sets.

5 Integrating Tools with a Marvel Environment

From the standpoint of an environment software development activities are performed by or through tools. In an open environment the appropriate tools for the development are added to the environment by the users. In the case of Marvel, and in fact with many other environments, the tools imported into the environment are raw Unix tools. To integrate a tool into a Marvel environment it is necessary to create an "envelope" which will allow the Marvel kernel to invoke the tool with the correct parameters and on exiting the tool will define the results of its having been used.

6 The Test Example

To test the viability of the overall software process modelling approach which Marvel exemplifies we attempted to build and use a Marvel environment for some example tools and build a test program in the environment we had constructed. The example, the Conic toolkit and a sample Conic application, is small but we feel illuminating.

6.1 The Conic Toolkit

Conic is a distributed systems development toolkit developed by the Distributed Software Engineering Group at Imperial College. It supports the construction of distributed and concurrent programs by supporting the development of individual software components and the building of systems from these components. It provides tools for compiling and debugging individual components and supports the creation, linking and execution of these components (Magee, Kramer & Sloman 1989).

Conic allows the definition of distributed systems by providing two languages: a programming language, for programming individual task modules (processes) with explicitly defined interfaces; and a configuration language, for the configuration of programs from groups of task modules. A variant of the configuration language is used to support dynamic creation, control and modification of these programs.

In Conic, context independent component types - group modules - can be defined within the configuration language and their interfaces given by typed entry- and exit-ports. Such component types may be built up from instances of other component types with the entry- and exit-ports linked to communicate by message passing. A system is a hierachical structure of component instances in which the bottom level consists of task module instances written in the Conic programming language and executing concurrently. Nodes define the mapping between the

component structure and the physical architecture. The programming language is Pascal extended to support message passing.

The Conic toolkit includes system development tools such as **cnc** (for invoking the compilers for both the configuration and programming language) and **ma** makefile generator (which uses the configuration language to establish dependencies). It also includes tools particularly appropriate for a distributed environment such as **pb** playback allowing the previous execution of a node to be replayed from a trace. To manage an executing program tools such as create, link and remove are used. The **iman** tool gives a common interface to these tool fragments.

Considerable experience has been built up in the construction of Conic programs and a design method based on this experience developed (Kramer, Magee & Finkelstein 1990).

6.2 A Conic Application

The test application selected is extremely simple, it passes a character “randomly” between a number of windows. The user determines the number of windows before running the application. When running the application the user enters in a starter window a character, a count and the number of the first window to receive the character and then the character is passed between the other windows until one window has displayed the character the determined number of times. The “result” is then displayed in the starter window. It consists of two component types, starter and window, defined by group modules starter.grp and window.grp. starter.grp uses the task module controller.tas and window.grp uses the task modules passer.tas and chooser.tas. All these modules use data type definitions held in a definition module windowinfo.def and chooser.tas also makes use of a library module to chose a “random” window number. All components (.tas, .grp and .def) are separately compiled. An executive is included into a group module to enable it to be a distributable, runnable node.

This application was constructed in both the “standard” Unix/Conic environment and reconstructed in the Marvel/Conic environment.

7 The Marvel/Conic Environment

The simple environment constructed supports the editing, compiling and building of a system comprising a number of nodes and executable script files. It also supports a limited form of dynamic change.

To build the environment it was necessary to define the data model, process model and tool envelopes. These were produced by incremental and exploratory development.

The software objects to be stored in the objectbase are node, executive group, task and definition modules. These are organised into an SCENVIRONMENT which associates Conic application programs with executive modules available for use within a distributed node. Each application

program is regarded as a SYSTEM. An example object definition is shown below:

```
# A node is the unit of execution and distribution, and is the same as a
# group module except that it must contain an executive i.e. it must use
# an executive group module

NODE :: superclass ENTITY;
    name                : string;
    availability_status  : (Available, NotAvailable) = NotAvailable;
    build_status         : (Built, NotBuilt) = NotBuilt;
    executive_used       : link EMODULE single;
    group_modules_used   : set_of GMODULE;
    task_modules_used    : set_of TMODULE;
    definitions_used     : set_of DMODULE;
    scripts              : set_of SCRIPT;   # Each script implicitly uses
end                                         # this node and the other
                                            # scripts of this node
```

The process model developed was straightforward: modules can be created or modified by editing; imported definition modules must be available in the same directory as the importing module or in a directory in the users search path; node modules can be compiled slightly differently from other modules, they may also be “built” by compiling all the node components from scratch. Because activities can be performed on many objects we were able to overload the rules somewhat.

An example process model rule is shown below:

```
# The rule for compiling a task module

compile[?t:TMODULE]:

# Precondition

(forall DMODULE ?du suchthat (member [?t.definitions_used ?du]))
:
(and(?du.compile_status = Compiled)
    (?t.compile_status = ToBeCompiled))

# Activity

{ COMPILER compile_module ?t }

# Postcondition

(?t.compile_status = Compiled);
(?t.compile_status = ToBeEdited);
```

The tool envelopes were Unix shell scripts. A typical such envelope, the details of which are not of particular interest, is shown below to give a flavour:

```
# This script provides the compiler envelope for a module
# usage: compile_module module

echo $0 $1 ...
compile_prog=cnc
cd $1
files_used=`ls *_used/*.*/*.* 2>/dev/null`
if [ "x$files_used" != "x" ]
then
    # copy all files associated with modules used by this module
    # into module directory
    cp *_used/*.*/*.* . 2>/dev/null
fi
module=`basename $1`
$compile_prog $module
if [ $? -eq 0 ]
then
    echo compile successful
    exit 0
else
    echo compile failed
    exit 1
fi
```

8 Lessons Learned

In trying to give an account of the lessons learned from the construction of the simple environment discussed above we have attempted to filter out the lessons which are too Marvel specific retaining those which we feel apply more generally to software process modelling as a whole.

8.1 Environment Construction

The rigid distinction between the data model and process model is clearly impossible to maintain. In setting out the objectbase schema the attributes of objects form the potential pre- and postconditions of the process model. The two are closely intertwined.

Mapping Conic development to the objectbase was relatively straightforward, this seems to have been so because Conic systems have a clear structural foundation. The relation between the underlying representation scheme and the objectbase schema (and hence the process model) is very important.

Many of the tools we use in software development are highly generic. That is, we use them many times over during the development process but in different circumstances. The model case of this is the editor. We see relatively few activities (where activity translates to tool invocation) with many pre- and postconditions.

Because the way in which process models are developed is necessarily incremental and exploratory there is a strong requirement for support for analysis and debugging. The “pure” process modelling aspects proved to be the least problematic aspect of constructing the environment.

The major problems we encountered were to do with building tool envelopes. We had tom ensure our tools were “well behaved” with respect to the data and process models. Clearly a major difficulty was the necessity of writing “hacky” shell scripts (with all that this implies). Further we had to embed significant knowledge about the process and data model in the envelopes and vice-versa. There had to be a match between the postconditions specified by the process model and the exit status delivered by the envelope hence further tying the envelopes and strategies together.

8.2 Environment Use

In general, using the Marvel/Conic environment was straightforward. However a problem was encountered in what might be termed “process over-automation”. Once an activity was invoked the user effectively lost control as the environment invoked further activities by chaining. It was, commonly, difficult to predict what activities might be invoked or their effects.

A specific difficulty arising from our use of the Marvel/Conic environment was data definition reuse. In contrast to C, Conic files cannot be compiled independently of external files which they reference. C files which are used by others need appear in only one position within the objectbase hierachy whereas common Conic files are required during the compilation of any file in which they are referenced. This proves to be very difficult to handle with Marvel. It might be argued that this problem is not fundamental but is simply due to a bug or lack of functionality with Marvel. Certainly Marvel could be patched to eliminate the difficulty. More significantly however it again points to the close interlinking of objectbase structure, process model and structural aspects of the representation scheme. Similar problems were encountered with handling of multiple instances of Conic components.

Setting the question of over-automation of the process aside, the use of the Marvel/Conic environment raises the issue of the granularity of the process model. In Marvel process control is at the level of tool invocation and interworking. Finer grain control could, in theory, be achieved but only if the behaviour of the tool was very well understood (unlikely in the case of a “foreign tool”) and intermediate interaction with the objectbase could be defined. It remains an open question as to whether the level of granularity provided by a Unix or similar toolset is, except in strictly limited circumstances, the appropriate level for automation. To do so is to reduce a model of programming to an invocation of vi!

In the final analysis we were left with the feeling that we had gained relatively little for the effort of building the Marvel/Conic environment. With complex and high functionality tools such as **cnc** and ma the additional integration provide by the Marvel infra-structure seemed to add little except graphical display of environment status.

9 Research Strategies

What are the implications of these observations for research strategy?

There has been a growing gap between research on the software development process and research on what might be called, for want of a better term, product representation schemes (specification languages and the like). Yet the concerns of the two strands of research have close parallels, for instance process modellers are concerned with coordinating the work of independent agents with differentiated roles and product specifiers with devising languages that provide for separation of concerns. This argues for research directed precisely at the point where process and product modelling are difficult to distinguish, in other words where the entities being manipulated are not anonymous “objects” but are meaning bearing elements of the underlying product language.

This argument also supports the need for fine-grain software development modelling by which we mean the analysis and description of the detailed structure and organisation of development activities. In general this structure and organisation is ignored by modelling software development at the level of tool invocation and interworking. Indeed we suggest that many important gross features of software development such as verification, validation and cooperation arise from the complex interplay of fine-grain activities. These features of software development are not simply embedded in a matrix of routine “house-keeping” tasks. Rather they are emergent properties that derive from the underlying fine-grain organisation. Fine-grain software development modelling may, as a by-product, build bridges between empirical (quantitative, experimental) studies of programmers behaviour and computational models.

The successful deployment of process modelling techniques is dependent upon what we have termed well behaved tools. This means that the underlying tool interfacing and encapsulation techniques must be developed. If we envisage a distributed setting this poses important research challenges for tool implementors and for systems and language designers.

As is shown by our example above, the use of process modelling within software development environments is closely tied to the architecture and tool integration strategy of that environment. We are strongly antipathetic to environments based on global databases or centralised control through broadcast. A preference for loosely coupled and distributable architectures based on tool linking and specialised file storage stems both from “software engineering instinct” and from experience with the difficulty of evolution and change in environments employing global integration schema. Further, we are strongly convinced that performance is of major importance in software development tools and environments. This again pushes us towards tool to tool integration (with direct transformations). How exactly software process modelling fits within such an architecture is uncertain.

There is always a fine balance to be struck in providing support for software development activities between automation on the one hand and allowing direct intervention by the developer. How precisely that balance is made is highly dependent on the activities under consideration. Nevertheless we are inclined to the view that much existing process modelling research swings too far in favour of automation. An alternative is to view a process model as a vehicle for providing guidance to the developer so that at any time the user could ask “what should I do next?" or perhaps “how do I get out of this mess?". In this setting the process model is more akin to the computer aided learning or tutoring than conventional software development support with all that this implies for research issues - what sort of model of the user needs to be preserved, how exactly should the guidance be presented. Further, it suggests that we look in general at

interfaces and support for tool control and management.

Experience with software process modelling brings home forcibly the point that a “generic” process modelling capability allows one to build “stupid” processes. To use these techniques appropriately it is important that we have some handle on what a “good” (economic, effective, uncertainty reducing) process would be.

Like any sort of “programming”, software process modelling is seductive. It is easy to get involved in the details and to set aside the question of whether or not the end product is of value. Most of the examples of software process modelling focus on configuration/version management, system building and test yet these are areas in which we already have powerful and effective tools (RCS/SCCS/Make and so on) albeit that software process modelling may provide more genericity in their support. If software process modelling is to demonstrate its value it must do so in areas in which there is an identifiable gap in development support, this means generally up-stream in design and specification and with rigorous approaches to development.

A by-product of the seductiveness of enactable software process modelling is the dominance of “environment” applications of software process modelling over its application in such areas as education (a traditional consumer of software process models) and process assessment both of which have received relatively little attention.

10 Conclusions

This paper has used Marvel and the Conic example as a framework for a critical analysis of software process modelling and has advanced an alternative (or at any rate complementary) research agenda in this area.

Many of the areas of research identified in this paper are also the subject of further work by the Marvel research team (Heineman, Kaiser, Barghouti & Ben-Shaul 1991, Barghouti & Kaiser 1991) and by others. A full review is beyond the scope of this paper but particular attention should be paid to the work of Feather, Fickas and Van Lamsweerde, for example Dardenne, Fickas & van Lamsweerde (1991).

A version of Marvel (3.0) is shortly to be released which provides support for multiple users. This version of Marvel, based on a client/server architecture, has a separate envelope language for providing interfaces to Marvel which is translated by a Marvel tool to sh, csh or ksh for execution. It also extends the objectbase implementation to support arbitrary relationships resulting in a directed graph rather than a tree. Other enhancements include tools for visual analysis of process models and built-in predicates to provide additional control on tool chaining.

The research agenda is the subject of further work by the authors (some preliminary results are given in Finkelstein, Kramer & Goedicke 1990). We intend to extend our evaluation of software process modelling particularly to examine upstream development and an extended tool set.

Acknowledgements

We wish to acknowledge the contribution of Gail Kaiser and her team at Columbia University to the work reported above. Although in no way responsible for the opinions we have expressed, by licensing Marvel for use by Universities they have made it possible. Marvel licenses are available

for research purposes to educational and other non-profit institutions for a small processing fee (contact Prof. Gail E. Kaiser, Columbia University, Department of Computer Science, 500 West 120th Street, New York, NY 10027, USA, kaiser@cs.columbia.edu). All criticisms of the overall approach set aside, Marvel is an interesting product providing a clear and well constructed demonstration of software process modelling techniques. It is a seminal contribution to the field and we would recommend it to any software engineering research group seeking to gain experience in this area.

Thanks also to our colleagues and students for their comments and technical assistance, in particular Jeff Magee, Manny Lehman and Kevin Twidle.

References

Barghouti N. & Kaiser G. (1991); Scaling Up Rule-based Environments; Columbia University Technical Report, CUCS-047-90.

Dardenne A., Fickas S. & van Lamsweerde A. (1991); Goal-directed Concept Acquisition in Requirements Elicitation; Proc. 6th International Workshop on Software Specification and Design; pp14-21, IEEE CS Press.

Dowson M. [Ed.] (1987); Iteration in the Software Process: Proceedings of the 3rd International Software Process Workshop; IEEE CS Press.

Dowson M. (1991); Manufacturing Complex Systems: Proceedings of the 1st International Conference on the Software Process; IEEE CS Press.

Finkelstein A., Kramer J. & Goedicke M. (1990); ViewPoint Oriented Software Development; Proc. of 3rd International Workshop Software Engineering & its Applications; Cigref EC2 V1, pp 337-351.

Heineman G., Kaiser G., Barghouti N. & Ben-Shaul I. (1991); Rule Chaining in Marvel; dynamic binding of parameters; 6th Annual Knowledge-based Software Engineering Conference; Rome Laboratory, New York, pp 276-287.

Kaiser G. & Feiler P. (1987); Intelligent Assistance Without Artificial Intelligence; 32nd IEEE Computer Society International Conference; PP236-241, IEEE CS Press.

Kaiser G., Barghouti N., Feiler, P. & Schwanke, R. (1988); Database Support for Knowledge-based Engineering Environments; IEEE Expert; V3, N3, pp 18-32.

Kaiser G., Feiler P. & Popovich S. (1988); Intelligent Assistance for Software Development and Maintenance; IEEE Software; V5 N3, pp 40-49.

Kramer J., Magee J. & Finkelstein A. (1990); A Constructive Approach to the Design of Distributed Systems; Proc. of 10th Int. Conf. on Distributed Computing Systems; May 1990, pp 580 - 587.

Magee J., Kramer J. & Sloman M. (1989); Constructing Distributed Systems in Conic; IEEE Transactions on Software Engineering; SE-15 (6).

Perry, D. [Ed.] (1990); Experience With Software Process Models: Proceedings of the 5th International Software Process Workshop; IEEE CS Press.

Potts C. [Ed.] (1984); Proceedings of the Software Process Workshop; IEEE CS Press.

Tully, C. [Ed] (1989); Representing and Enacting the Software Process: Proceedings of the 4th International Software Process Workshop; ACM SIGSOFT Software Engineering Notes; V14, N4.

Wile, D. (1983); Program Developments: formal explanations of implementations; CACM 26(11), pp 902-911.

Wileden J. & Dowson M. [Eds.] (1986); Software Process and Software Environments:Proceedings of the 2nd International Software Process Workshop; ACM SIGSOFT Software Engineering Notes; V11, N4.