Process management and the systems development life cycle

R. Tolido

Cap Volmac, Dalton Iaan 300, Postbus 2575, 3500 GN Utrecht, The Netherlands

ABSTRACT

The purpose of this paper is to describe certain aspects of modern systems development life cycles - such as iterative development, re-use and participatory design - and the way process modelling techniques can be applied to them. This leads to a definition and description of enactment software based on such techniques.

Traditional 'paper based' approaches (possibly automated by tools) to the systems development life cycle are proving to be inadequate to structure and manage complex projects in which many people are involved, changes occur very frequently and in which a lot of fine-grained, inter-related activities need to be carried out, possibly in an iterative fashion. An appropriate method, covering project management, quality management and development is essential. Such a method is briefly described in this paper. An assessment is made of the aspects of this method that clearly indicate the need of automated tools support.

To provide this kind of support successfully, techniques are described to define well formed life cycles in terms of the structure of the development process and the dynamics of the interaction of process, people and development tools. These modelling techniques can be articulated in process management software which provides the 'method engineer' with the facilities to define generic models, the project manager with facilities to tune the generic models to the specific needs of the project and finally to run the project model as a long running program which guides and controls all development effort. A short example is given of the way this process management software can be linked to other tools in order to provide advanced, automated support for parts of a complex life cycle.
In conclusion, the paper stipulates two points. First, true quality management of modern, complex systems development can only be achieved with software that is capable of modelling and managing the systems development life cycle and the integration of all of its constituent components. Secondly, the availability of this kind of software may itself be the 'IT-enabled' trigger to consider new approaches to systems development.

1. INTRODUCTION

1.1 Frequent change Over the recent past, the (business) world has become more and more volatile. Markets are changing rapidly, as is legislation. Speed-of-adaption and time-to-market can, more than any other factors, determine an organisation's competitiveness and therefore success and survival.

This rapid and continuing change is driving many organisations to re-engineer their business, in terms of structure, processes and people. Information Technology plays an increasingly important role within this context of Business Process Redesign.

Of course, information systems must be able to support the frequent changes that occur within the organisation. When a new product is rapidly developed and made ready for exposure to the market, changes or adaptations to the supporting information systems must be applied at least with the same speed. This also pertains to changes to the organisational structure itself: information systems must, for example, be able to accommodate effectively the splitting or merging of tasks and the shifting of responsibilities of players within a certain business process.

On the other hand the availability of new IT products, such as Workflow Management Software and distributed system environments, can be the very reason to re-engineer business processes. IT is in that case the driving force for structuring the organisation in a way that could never have been considered without IT.

2. CONVENTIONAL APPROACHES TO SYSTEMS DEVELOPMENT

Conventional systems development methods are not always suitable to apply to the dynamic, frequently changing environments of today. This is not a new statement: many publications have been addressing this issue, notably [Boehm86]. The problem is therefore only briefly discussed in this paper.

First, conventional methods are usually based on a linear sequence of phases and activities to be carried out. Each phase or activity cannot be started before the preceding one has been completed. This implies a long process for specifying an information systems in its entirety. The transition
of one phase to the other is usually based on large milestone documents that need to be verified and validated. This leads to huge amounts of paper, together with the inherent gap between systems developers and users, the latter often not fully able to grasp the meaning of the modelling techniques the systems developers have been using.

Furthermore, a milestone document is frequently regarded as a disposable (or at best non-essential) one: systems developers categorise themselves according to the phases a life cycle consists of: information analysts, functional designers, technical designers, programmers, testers, etc. The milestone document thus also becomes the main medium for troublesome, possibly erroneous communication between systems developers.

Secondly, conventional approaches are not appropriate to deal with complex and constantly evolving requirements. This is mainly due to the linear structure of these approaches, in which the actual software only appears at a very late state in the systems development process. The troublesome 'paper based’ communication between systems developers and users leads to too little user-involvement, ultimately resulting in misunderstanding and disappointment. When an information system has finally been built, the initial requirements more and more turn out to be outdated the moment the system is actually to be implemented in the organisation.

Experience thus shows that the resulting systems do not meet user needs and expectations, show a distinct lack of flexibility and therefore cause severe maintenance problems once implemented.

3. ITERATIVE DEVELOPMENT

The method that is described in this section, PERFORM Iterative Application Development (IAD) (Figure 1), is the approach the Cap Gemini Sogeti group has developed to deal effectively with the changed
environment in which systems developers increasingly have to operate. \textit{PERFORM} IAD ([Cap94]) is not revolutionary. It can best be described as a synthesis of successful project principles and new technology applied in a suitable way to produce a faster, more flexible, more relevant approach to the development of modern information systems. \textit{PERFORM} IAD is built around an iterative life cycle, making use of new development techniques (such as object-orientation and client/server design) and tools. It also provides a managerial approach to reuse and focuses explicitly on quality throughout the life cycle. Rapid, iterative application development clearly has been described in many publications, notably in [Gilb88] and [Martin91], but not always in the same amount of detail and not always covering the full scope of the systems development life cycle. \textit{PERFORM} IAD is based on the following fundamental concepts:

- \textbf{Iterative life cycle} The life cycle consists of a small number of phases that are executed in an iterative way. Three main options are available (see the diagram), depending upon criteria such as complexity or stability of the requirements and the need for quick implementation of parts of the system.

  In every case, a \textit{Pilot} is delivered (and possibly implemented) at the end of each iteration, with the objective of meeting a jointly agreed subset of requirements. Typically, a full iteration may only take two months or less. Experiences acquired with building and implementing previous pilots clearly influence the specification and development of subsequent pilots.

- \textbf{Joint Application Development} Frequent use is made of workshops throughout the life cycle. Systems Developers and users jointly reach consensus on the development strategy, requirements, specifications and the design, thus striving for interactive validation. They also together review, validate and test 'in between' versions of the system under construction. Workshops lead to a faster development cycle, realistic specifications, fewer milestone deliverables and less paper based validation activities. Different types of workshops are conducted throughout the life cycle, in order to encourage team-working and facilitate real-time decisions making:
  - Joint Development Strategy
  - Joint Requirements Definition
  - Joint Pilot Design
  - Joint Review and Testing
  - Joint Feedback Collection
  - Prototyping

  During workshops and development tasks, prototyping tools are used extensively to build different types of prototypes, depending on the initial objectives. The types of prototypes may include \textit{exploratory} prototypes, to test or prove the feasibility of a certain aspect of the system, \textit{evolutionary} prototypes -actually pre-releases of the pilot being developed-
meant to generate feedback during development and operational prototypes, which consist of specific enhancements build on top of an implemented pilot in order to investigate new requirements.

- **Re-use** The method propagates frequent re-use or acquisition of existing components throughout the life cycle. In practice, every task description of the PERFORM IAD method contains a separate subsection on re-use, describing what measures should be taken *before* the task is being carried out (searching for components that can be re-used) and *after* (making sure that components that have been created can be re-used in later occasions). Re-use is essential to iterative development. First, because a pilot can only be built quickly when it can be based as much as possible on existing components. Secondly, because the quality of a re-usable component can be considered unassailable; quality-related activities can thus be directed completely to the parts of the pilot that are new, which greatly saves time and effort.

- **Quality** Analogous to re-use, quality aspects are covered throughout the life cycle. Each task description contains a separate section 'validation and verification' which exactly describes what quality constraints pertain to the specific task, again covering what constraints should be met *before* the task is actually carried out and what constraints pertain before ending it.

- **Cooperative development** A pilot is developed by one or more A-teams, possibly working in parallel. An A-team is a small team of responsible and skilled people with a high degree of synergy. Between them they have the necessary portfolio of skills required to analyze, design, build and implement the pilot. An A-team does not have clear role definitions for the individuals (such as team leaders, systems designers, programmers), but rather contains individuals who cover many of the skills required during systems development and actively practice all of these skills.

  A-teams are crucial to iterative development. The appearance of A-teams follow the example set by engineering where the direction has been away from an assembly line (or serial approach) to concurrent engineering, using small teams, each covering the entire engineering process. As the system is built and delivered in small components it is essential that there is a high degree of communication between team members to ensure the integrity of each component and the final system. Using a small, responsive team overcomes the problem of instructions or documentation being misinterpreted as they are passed between different groups of people.
4. PROBLEMS WITH MANAGING THE ITERATIVE LIFE CYCLE

Some problems pertaining to the management of iterative life cycles such as *PERFORM IAD* are:

- **Re-use and software configuration management**  On a typical project, software configuration management is based on controlling changes to items. Usually, these software configuration items (SCI’s) have been associated with entire subsystems. However, in an environment that relies heavily on re-using relatively small components (in object-oriented development a component will typically be a class), each component has a separate life. A re-usable component is not particularly associated with one application or subsystem and must therefore be considered a separate software configuration item. Since each component may be used in many different cases, it will be extraordinarily difficult to keep track of dependencies. A change or a fix to a component can thus not be managed or promulgated. Furthermore, re-usable components are likely to be tailored for use in specific applications or subsystems. Control procedures to handle multiple authors of the same component will have to be defined.

- **Iterative development and software configuration management**  Configuration control is a discipline that is challenged to its limits when applying an iterative method that is based on short cycles and multiple teams, possibly working in parallel on one or more pilots. It is unavoidable that, at one time, there will be multiple versions of all baselines deployed, multiple new pilots under development and multiple teams of developers. Changes must be planned, orchestrated and controlled. All parties interested in those changes, all of whom may possess disparate goals, must be coordinated and informed in timely fashion.

- **Frequent changes to requirements and specifications**  Iterative development aims at maximising the feedback from earlier pilots when specifying a new pilot. A system gradually 'evolves' to a more or less final state, going through multiple iterations of the life cycle. Experiences with earlier pilots will result in many changes, enhancements or entirely new specifications. In such an environment it is hard to keep track of all proposed changes. Many people may be involved in processing a change proposal, each having their own role in the process.

- **Iterative estimates**  Iterative development, and the feedback it produces, often lead to management acceptance of revised estimates for time and money, notably at the early stages of the project. The advantage is that it removes, to some extent, unpleasant and unreasonable pressure on the developers. In general, the iterative approach leads to dynamic planning.
Based on a growing amount of experience, acquired in previous iterations, estimates are constantly being improved during the many development steps. As a consequence, management can and will have justifiable confidence in such estimate requirements. The drawback is that estimates must be made much more frequently than in the past. Furthermore, their accuracy is definitely presumed to improve over time, in order not to brutalise this valuable confidence. The challenge is to utilise as much as possible project attributes and estimates from previous iterations, to produce reliable new estimates. These attributes and estimates should be carefully logged, always at exactly the same stage in the development process. In this way, the metrics of different iterations can safely be compared to each other when making estimates.

- Other ways of verification Verification is usually a very time-consuming process, because lots of connections between many components must be checked for traceability and completeness. This is all the more true when applying iterative development, in which specifications can very easily and frequently be changed, thus leading to the need for frequent re-verification.

If verification is still to be carried out manually, the process may be unsuitable for inclusion in joint workshops. This would be an undesirable situation, undermining the whole concept.

It is clear that iterative development introduces some new, unprecedented challenges to managing the development process. As with many complex processes, automated support may provide some ways to face these challenges. To provide for such support, relevant parts of the process must be modelled using structured, unambiguous formalisms. Once these models exist, they may be expressed, executed and controlled using automated tools. This is the subject of the next section.

5. TECHNIQUES AND TOOLS FOR PROCESS MODELLING

5.1 Techniques Describing (systems development) processes to a level that is appropriate for automated support, requires special formalisms:

- Notational techniques must be available to model process decomposition. This has been common practice in software engineering for a long time; it is a good way to model a process at various levels of detail, thus minimising the complexity at any level. Many modelling conventions for process decomposition currently exist.

- A technique must be available to describe the dynamics of a process at any level of detail: the sequence of activities within a process must be properly modelled, the states in which a process can occur and the conditions and actions that apply to the transition from one state to another. Since a modern, iterative life cycle will also provide for parallel
development, this aspect must also be covered by the technique, including facilities for synchronisation.

- Especially for iterative life cycles, where collaborative development implies very frequent, fine-grained communication between several people involved, a mechanism must be available to model this type of communication.

5.2 A tool for process modelling and management  In this section, a tool is described that is used to address some of the process management issues that have been raised so far. The SDW Process Modeller & Manager (SDW-PMM) is a part of PERFORMance, a tool set that has been defined to provide automated support for parts of the PERFORM quality system. SDW-PMM consists of components to define process models -in various levels of detail, both at the company and project level- and components to actually instantiate a process model, controlling the execution of it.

SDW-PMM is largely based on the concepts of two similar products that have been developed within the Cap Gemini Sogeti group: the Unix tool Process Weaver (described in [Cap92] and many other publications) and Project PROWESS, which has been developed by Hoskyns UK. SDW-PMM has been further expanded towards a tight integration with the PERFORMance CASE repository. As can be seen from two examples in the next section, this integration provides some excellent opportunities to combine SDW-PMM with other tools, in this case related to Software Configuration Management and estimation.

Clearly, with the description of the tool, a set of techniques for process modelling is implicitly described.

SDW-PMM consists, not surprisingly, of a 'modelling' and a 'managing' part. The modelling subsystem consists of two more or less identical modules to define processes respectively at the organisational and project level, models on the latter level being further specialisations of more general models at the organisational level. For the purpose of this paper, only the organisational modelling tool will be described.

The managing subsystem consists of an 'engine' to instantiate a process model and control its execution and an 'agenda', being the main means of communication between SDW-PMM and its users once it is executed.

5.3 The Process Modelling module  A process model can be defined in the process modelling module, using three main types of editors:

- Activity Diagram Editor
- Cooperative Procedure Diagram Editor
- Work Context Editor
1.1. Initiate problem definition

1.2. Study problem background

1.3. Analyse current system

1.3.1. Define Equipment and Staffing

1.3.2. Prepare Model of Current System

1.3.3. Describe Outline Business Context

1.3.4. Describe Interf. with Other Sys.

1.3.5. Describe Workload

1.3.6. Prepare Problem Definition

1.4. Define staff operation costs

1.5. Plan feasibility study

1.6. Report & Review

### Figure 2  Example of an Activity Diagram

#### 5.3.1. The Activity Diagram Editor
A process model is first described, on the highest level, as an organised set of activities. As an example, a process model describing a 'Business Study' method, would contain a Problem Definition activity (Figure 2) and a Feasibility Study activity. Each of these activities can be subdivided into smaller activities, and so on. Activities that have no further decomposition are called 'terminal'. Each activity is described in the repository in terms of input/output components, the roles (types/groups of people) that are authorised to perform them.

An activity diagram only provides a hierarchical view of activities. Each activity - on each level - is linked to a cooperative procedure, which expresses the dynamics of its sub-activities. At the lowest level, the terminal activities and their associated cooperative procedures control the interactions with the final users.

#### 5.3.2. The cooperative procedure diagram editor
A procedure diagram thus exists for each activity in order to describe its dynamics. Cooperative procedures are based on the well known Petri Net formalism ([Bra86]) with enhancements to model certain advanced process dynamics and inter-user communication.

A cooperative procedure is a directed graph, containing two types of nodes, **places** and **transitions**, and the links between them. A place can be linked to one or more transitions, and vice versa, but a place can neither be linked to another place nor a transition to another transition. The places in a cooperative procedure represent the possible states of the procedure. At any given time, a certain number of these places are marked, containing a token. The set of places that are thus marked represent the current state of the entire cooperative procedure.
The state of the cooperative procedure can change as a result of external events. When an event has occurred, all the transitions of which all immediately preceding places are marked, are considered. A transition is made up of two parts: a condition and an action. Whenever a transition is considered, and its condition is true, its action is executed and the token that was in the place preceding the transition passes to the place following the transition.

Several links can be defined between places and transitions, including one-to-one, one-to-many and many-to-one, each having their own semantics when modelling a cooperative procedure.

The most basic configuration is that of one place linked to one transition, followed by one place (Figure 3) (the following graphical convention is adopted: a place is denoted by a circle, a transition is denoted by two stacked rectangles. The links are shown by arrows). This basic configuration can be used to model sequential execution of conditions and actions, one step following another, with no possibilities of branching.

Then, a single place can be followed by different transitions (Figure 4). This corresponds to a 'case' construct, in which normally one of the conditions should be equal to 'true' and corresponding actions should be executed.

An analogous configuration exists for the link from one transition to many places (Figure 5). If the condition part of this transition equals 'true', new tokens are created so that there is one for each place following the transaction. This results in the creation of parallel threads of execution in the procedure.

Each thread will then continue execution independently.

As concluded, there are moments in parallel procedures that need synchronisation of different threads. In that case, the many-place-to-one-transition construct can be utilised. It was already indicated that, before a transition is considered, all of its

---

**Figure 3**

One-to-one link

**Figure 4**

One place to many transitions

**Figure 5**

One transition to many places
preceding places must be marked with a token. Consequently, if one thread already reached a place and others have not, it will simply wait until other threads have caught up. As shown in the figure, there will only be one thread of execution after traversing the transition. Of course, if the parallelism is to continue, several places can be linked to the transition.

The final configuration, consisting of many transitions linked to one place, (Figure 6) can serve different purposes. This depends on how the split that gave rise to the multiple incoming transitions occurred. If the split was the result of a passage from one place to many transitions, the construct typically serves as an 'end case'. On the other hand, if the split was the result of a passage from one transition to many places (indicating parallelism), this construct functions as a logical 'or': the place can be reached from each of the transitions.

As a small example, consider a cooperative procedure for developing a certain piece of software. The procedure is composed of two phases, an initial coding phase, followed by a quality review phase. Two programmers (A and B) will be working in parallel on different parts of the software. When they are both done - but not before that - the program will be handed over to two quality reviewers (C and D) who will both independently review the software. The software will be considered ready if at least one of the reviewers actually has performed the necessary review.

A cooperative procedure that models such an activity, would consist of the following elements:
- The initial place is marked 'start'
- The transition is directly made to 'assign work', since the condition part of this transition is not filled in. However, in the action part work is sent to the programmers A and B (more about how to model that in the next section). The two programmers will now work in parallel. Once they have both satisfied the condition (work is done) to the next transition, a shift will be made to a transition - again without a condition part - in which the program is sent to both reviewers.
- When either C or D is done, a transition is made to the final place 'end', which completes this particular cooperative procedure (Figure 7).

Clearly, this cooperative procedure can easily be enhanced, for example to deal with the iteration that may occur if the review does not succeed and some of the programming work needs to be redone.
Furthermore, as will be demonstrated in the next section, other possibilities exist to synchronise certain parallel activities. Using them might have simplified the cooperative procedure even more.

5.3.3 Elaborating on conditions and actions Once a transition has been defined (as a named combination of a condition and an action), both the condition and action can be further described.

A set of icons is available to denote different types of conditions (Figure 8):

- **Event** This condition waits for a certain event to occur. This event may have been raised by any tool that has been connected to the repository that SDW-PMM is also connected to. The event can also be expressed as a logical expression using SDWrite, a procedural (Unix shell-like) language to manipulate the repository contents.

- **Wctxt (work-context)** This condition waits for a work-context that has been previously sent, using the 'action' part of a transition. The work-context is the primary means of communication when using SDW-PMM. It will be explained in some further detail in the next section. Again, a logical expression can be used to constrain further when the condition is actually 'true'.
- **Collect** This condition collects answers to work-contexts, previously sent to a group of people. The same mechanism can also be used to collect multiple events. This time, two logical expressions can be described. The first, if true, will lead to the execution of a SDWrite-procedure (also to be described when modelling the collect-condition). This is repeated for every work-context that is sent back or for every event that has been recognised. The second expression, if true, will have the entire expression equal to 'true'. Note that this type of condition might have simplified the example cooperative procedure in the previous section.

- **Proc (procedure)** This condition is used to express that the cooperative procedure waits for a given place to be reached by a cooperative procedure that has been launched before (as will be seen, this launching can be done from the 'action' part of the transition).

- **Empty** This condition is used in these cases where no particular condition applies.

In analogy to this, five types of action can be described (Figure 9):

- **Wctxt (work-context)** This action sends a work-context to a person, known to the system. Before the work-context is launched, a SDWrite-script can be executed. The results of the script may be used to indicate to whom the work-context must be sent.

- **Progr (programming)** This action is used to execute certain types of programs, including SDWrite-scripts, interactive input-forms and external applications.

- **Distr (distributing)** This action sends a work-context to a group of participants.

- **Proc (procedure)** This action will launch a new cooperative procedure which will be considered a child of the cooperative procedure that launched it.
- Empty  No action will be performed.

The combination of the described graphical formalisms, combined with the expression power of the SDWrite script language provides sufficient facilities to describe cooperative procedures of even considerable complexity.

5.3.4 The work-context concept  One concept must be described that is a crucial addition to the modelling techniques that have been described so far: the work-context. A work-context model, which can be defined using a separate work-context editor, is a representation of the working environment of a user when performing a certain task. It contains a textual description of the work to be performed, and a command part. This command part consists of (deliverable) components that need to be queried, created or modified when performing the task. For each component, one or more tools can be described that can be launched to manipulate the component. Furthermore, a work-context can contain several buttons that can be used to formulate an answer to the sender of the work-context.

As an example, a work-context model to describe a conceptual data model may consist of a component that contains a list of relevant requirements (to be queried using an associated matrix-editor) a component representing the actual entity relationship diagram (associated with a CASE-tool editor to modify it), a component to hold a list of functional test specifications (associated with the specification editor of a test tool) and a component to hold any design decision that has been made during the process (simply linked to an external text editor). Furthermore, when defining a work-context model it can be described to which users a work-context instance actually can be sent.

A work-context instance may be instantiated from a model, populated with the right information (for example 'requirements list REQ99.2', 'ERD Customers-Orders', 'REQ992.tst' and 'remarks.erd') and sent to a specific user. As has been shown, a work-context instance is usually sent in the 'action' part of a transition. It may however in some cases also be sent 'manually'.

The work-context turns out to be a very efficient way of communication in a highly cooperative development environment. To the sender, it provides the means to express clearly what work needs to be done, including the components that need to be manipulated, visualised through icons, and additional help information. To the receiver, a work-context consists of everything that is needed to perform the task, including pre-defined links to tools that are to be used and mechanisms to signal the sender about the status of the task.
5.4 The Agenda A separate tool, the SDW-PMM Agenda, is used as the primary interface to SDW-PMM, once a process model has been launched and is executing. The agenda gives easy access to product components and their associated tools, presenting the user with a clear view of the tasks that need to be carried out (Figure 10).

The agenda provides the following functionality:
- receiving work-contexts, all of them being shown in a main window,
- loading work contexts that have been received,
- accessing components, tools and information that are bundled in a work-context in order to accomplish a specific activity,
- sending or forwarding work-contexts to other people,
- running complete cooperative procedures,
- scheduling the loading or running of a work-context at a given date or periodically

Once a work-context is activated by the receiving user, it will be presented in a dialog with all additional information, components that need to be manipulated, tools that can be launched to support that and buttons to notify the sender (Figure 11).

6. APPLICATION OF PROCESS MODELLING AND MANAGEMENT

6.1 Introduction In this section, a short example is given of the way an identified process management problem can be tackled, using a combination of process modelling, SDW-PMM and other tools. It leads to automated support for an advanced approach to (iterative) estimating.

At the conclusion of this section, a summary is given of some other areas in which process modelling and management can effectively be applied.
6.2 Iterative estimating  As stated earlier in this paper, estimating is a difficult discipline, especially in the context of iterative development: estimates must be made over and over again and should, after a few iterations, become more reliable. Existing approaches as Function Point Analysis do not always suffice in an iterative context, that often involves the development of event-driven, GUI-based applications in a client/server architecture and the application of object-oriented design techniques.

During the ESPRIT project MERMAID ([Mer92^]) in-depth research has been carried out on new, advanced ways of estimation. The result was a method that consists of several new estimation approaches, among them analogy based estimating and statistical based estimating. The first approach strives, when estimating for a project, to find 'similar' projects that have been carried out before. The second approach is based on a statistical foundation, trying to find a correlation between the attributes of earlier projects and its costs. When an estimate is to be made for a new project, this correlation can be used to predict costs, given the fact that some influential attributes have already been captured. These approaches seem very promising, but there are a few constraints to their practical usefulness:

- The attributes of a project (any attribute that might influence the estimate) must all carefully be logged. Because the exact influence of an attribute is not always clear (let alone the question whether it has any influence anyway), many attributes may need to be logged.
- The same attributes must be logged for every iteration -or project- in order to be able to compare iterations or projects effectively.
- Attributes must always be logged at exactly the same milestone in the life cycle (e.g. the end of a phase or activity or task), because its value may otherwise be questionable and thus unreliable.

The Cap Volmac tool ESTEEM is a spin-off of the MERMAID project. It supports statistical based estimating, providing interfaces to import project attributes, perform the actual estimation and export its results. Many of the project attributes to be logged can be derived from external tools such as SDW (for analysis and design) and PMW (project actuals). Interface software has been developed to extract the necessary information from these tools.

Process Modelling techniques and tools can effectively be applied to tie all these components together. It provides the automated support to build and maintain a reliable repository of project attributes. This repository can then be used as a basis for new estimates. At all milestones in the project life cycle (end of Requirements Definition, end of Pilot Design workshop, end of Pilot construction), the corresponding parts in the process model are enhanced with procedures that automatically launch the proper interface utilities. These utilities extract the relevant project attributes from external tools. Then, these attributes are uploaded into the
ESTEEM repository. No manual intervention is needed. Of course, a work-context could be sent to a reviewer to double check the feasibility of the extracted attributes.

Worth mentioning is that, in order to integrate ESTEEM successfully with process management software, ESTEEM's external interface had to be entirely re-specified and built. The software was not prepared to be embedded in the subtle framework of SDW-PMM, which typically takes over most of the high-level menu-handling of the applications it connects to. ESTEEM had to be redeveloped as a set of small, loosely-coupled modules that could each be interfaced to as a separate unit (Figure 12).

6.3 Other areas of application Many other areas, related to modern, iterative systems development, show promising characteristics for the successful application of process modelling and management tools. To name a few:

- **Quality assurance** Review and testing sessions can automatically be triggered when a component has been announced ready. Next activities will only be executed when reviews have been passed successfully. Some systems development methodologies have standard quality procedures that are to be performed after finishing each activity. These procedures can be modelled as standard cooperative procedures and can thus be linked to any other part of the process model.

  Furthermore, many verification checks - for example checking the mutual consistency between analysis and design models - can automatically be called as part of a condition to shift to a new activity.

  The availability of repository-based development tools can be a great support when dealing with verification in an iterative context. The meta-model of the repository must be enhanced, so that all components that are created during the life cycle can be stored, including the associations they have with other components. Verification requirements must be expressed as rules in the repository. They can then indeed be automatically executed by the process management software at all times. Frequently modifying parts of the system under development is now possible, because the impact of a modification on the rest of the system can quickly be analysed and dealt with.
- **Re-use** Frequent re-use at all stages of development involves a lot of communication between A-teams, coordinating 'class librarians' and quality assurance personnel. This tight network of inter-related activities can very efficiently be modelled and managed through a set of dedicated cooperative procedures. Among other things, this also will imply the proper, automatic notification of changes to all developers (or teams) that have 'subscribed' to certain re-usable components.

- **Software Configuration Management** Currently, some tools exist to define and maintain the static aspects of an advanced software configuration management system. This includes tools to store and retrieve multiple baselines and versions of software configuration items, but also tools to keep track of change requests, releases and versions (possibly overlapping) and tools to perform detailed impact analysis. Process modelling and management tools can be brought into action to navigate through this incredible amount of diverse information. Such an approach will almost be inevitable to control the huge number of inter-related activities, involving many people from many disciplines, when applying iterative development.

- **Cooperative development** The striving for team-based, cooperative development raises some additional challenges to process modelling and management. A well-coordinated team working in parallel, can for example rely on an optimistic concurrency-control strategy. Parallel development on the same components might in that case be feasible, as long as everybody is constantly aware of the activities of their fellow team-members (thus implicitly avoiding potential clashes). Any relevant events - changes to components, newly created documents, important design decisions - must therefore be automatically propagated to everybody involved.

An interesting form of cooperative development is that of 'structured conversation' ([Flores88]) in which many people can participate in a communication-network, for example to reach important design decisions through consensus. Automated support can be very useful, for example to keep track of all discussions a user is currently involved in and to make sure contributions to discussions are supplied in time.

An additional challenge in this context is how to involve users, who should participate fully in the iterative development process.

- **(Re)designing the development process** Many present-day systems development methods show an explicit, built-in flexibility. An organisation can compose its own methodological framework, selecting only components (activities, deliverables, etc.) that pertain to the problem domain of the organisation. After that, a project leader can even further tailor the composed method, creating a 'scenario', to have it fit precisely to his current project context. In both cases, a set of process modelling tools -
such as described in this paper- will be of great help. The 'method engineer' or project leader is not likely to experiment with different scenarios if this is a troublesome, lengthy exercise. On the other hand, if he can have a library of re-usable, methodological components at his immediate disposal, this will encourage him to strive for the most appropriate mix.

Another aspect of a flexible method is that all of its peculiarities should be subject to frequent re-assessment, aiming for continuous improvement of the development process itself. A method such as PERFORM IAD contains several activities at the beginning and the end of each iteration to perform this re-assessment. Of course, the supporting process modelling software must accommodate resulting changes to the development process. Furthermore, the 'managing' component of process modelling and management software can provide for very useful statistics on the course of the development process. Statistical information on for example the elapsed time of activities and their synchronisation can be ideal input to locate bottle-necks or inefficient processing, both in relationship to the method and the resources involved.

Finally: the formalisms that have been introduced in this paper to model and manage complex processes, largely originate from simulation (e.g. Petri Nets). This clearly suggests that simulation would be an interesting approach for improving the method, both when initially defining it and later, during re-assessment.

CONCLUSION

It has been shown that modern approaches to systems development introduce new, unequalled challenges to managing the development process. Among other things, this is due to factors as the lack of milestone documents, very frequent changes on all levels, speed of development, cooperative team-development, strong user-involvement and complex relationships between product components.

In order to guarantee the appropriate level of quality, relevant aspects of the life cycle must be modelled using structured, unambiguous formalisms. Once these models exist, they can be expressed using automated tools. Such a process modelling tool, consisting of several graphical and textual editors, has been presented. When the models have been defined in the tool repository, they can be instantiated, thus providing automated support for the actual execution and control of the modelled processes. The result is an integrating framework that may even constitute all components of the systems development life cycle. A short example of the application of such a framework has been given, together with the description of some other interesting areas that apply.
As a final remark: it must be realised that the process management concepts that have been described are not revolutionary at all. Actually, *workflow management software* is gaining momentum as the driving force for structuring and improving many kinds of business process. *Business Process Redesign* aims at optimising processes, fully inspired by the possibilities of modern information technology. This does not seem to be an unreasonable approach to apply to the systems development process either. Using the potential of automated tools support in composing a new systems development approach is a process whose time has come.
REFERENCES