Change Analysis and Management in a Reuse-Oriented Software Development Setting

Wing Lam

Department of Computer Science, University of Hertfordshire
College Lane, Hatfield, Herts AL10 9AB, UK
W.Lam@herts.ac.uk, Phone: +44 (0)1707 284337, Fax: +44 (0)1707 284303

Abstract. This paper discusses the need for systematic and methodical approaches for change analysis and management in software projects. We are *not* concerned with specific techniques (such as program slicing or dependency graphs), but with the overall project-level process of handling change, which we feel is under-represented in the current literature. The paper describes the efforts being made to manage change at Software Development Services (SDS), a small commercial organisation that specialises in the development of Customer Complaint Systems (CCSs). Because SDS recognises that most change is specific to their domain (CCSs), we are taking a *domain-specific* approach to change analysis and management. Our main contribution in this paper is a framework for the change process, called the *Change Cycle*, which is concerned with identifying and formalising reusable change knowledge. The paper reviews current methods and techniques for handling change, and discusses the desirable characteristics of a change process. We then describe the Change Cycle in detail and our experience of using it at SDS. Tool support for change is also outlined, as is our approach to evaluating this work. We conclude that change analysis and management should be treated as an integral part of the software process, not as an adjunct to it.

1. The Need for Systematic Approaches for Handling Change

Software systems are not static, but evolve. Fluctuating user requirements, commercial pressures, organisational transition and demands for interoperability (e.g. the Internet) all contribute to volatility in the software process [14, 18]. As [16] noted, *"software systems must change and adapt to the environment, or become progressively less useful"*. Increasingly then, today's software engineers need systematic approaches for dealing with change.

This paper describes the efforts being made to manage change at Software Development Services (SDS), a small commercial organisation than specialises in the development of Customer Complaint Systems (CCSs). Because CCSs are similar in functionality, SDS has adopted a reuse-oriented software process, where individual CCSs are tailored from a generic CCS prototype. SDS has observed that many of the kinds of changes made to one CCS are similar to the kinds of changes made to other CCSs, i.e. much change is *domain-specific*. This paper describes how we attempted to exploit this observation in developing a systematic approach to the analysis and management of change at SDS.

The format for this paper is as follows. The next section, Section 2, describes the problem of change from the perspective of SDS. Section 3 reviews current methods

for handling change in the literature and indicates how the approach discussed here differs. Section 4 discusses the desirable characteristics of a change process. Section 5 presents our framework for change analysis which we call the 'change cycle'. Section 6 describes the application of the change cycle to change problems at SDS. Section 7 outlines tool support for change and Section 8 the strategy we are using to evaluate our work. Finally, Section 9 concludes.

2. Change in a Reuse-Oriented Software Process

SDS was formed in 1994 to meet the growing market for Customer Complaint Systems (CCSs). The majority of clients for CCSs are retail outlets. In short, a CCS records individual customer complaints in a desired format, tracks the complaint from initiation through to resolution and performs a variety of anlayses over complaint data (e.g. % of complaints resolved within a given time, distribution of complaints relating to a defined service or product group). Like many commercial applications today, CCSs are typically required to run under the Windows environment, either as a stand-alone or networked (multi-user) application.

Because CCSs tend to share similar functionality, SDS has adopted a reuse-oriented software process, where an individual CCS for a customer is tailored from a generic CCS prototype. In the SDS software process, change is seen as a central part of the software process of which there are three main categories (Figure 1).

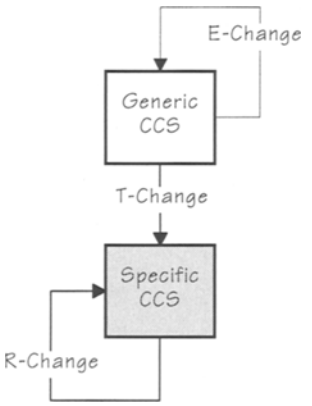

Figure 1 Types of change

1. *Tailoring changes (T-Changes).* This is where SDS identifies with the customer the changes needed to tailor the generic CCS to their specific CCS requirements.
2. *Requirement changes (R-Changes).* In addition, like many software systems, changes (which are customer-driven) also occurs during the development process (e.g. demonstrations, trials) and after a CCS has been installed on-site.
3. *Evolutionary changes (E-Changes).* The generic CCS is itself the subject of evolutionary improvements. As SDS develops more CCSs, we expect the generic CCS to gradually become more 'mature'.

This paper is concerned primarily with *R-Change*, which [9] identify as a major challenge to software engineering (use of the word 'change' from this point on will refer to R-change). Like many 'real world' applications, changing and evolving requirements is a prominent feature of CCS software development projects. As CCSs are highly interactive systems, SDS use a user-centred and prototyping approach to their development. While such an approach is useful in 'discovering' customer requirements, we have found that there is little or no control over the change process itself.

SDS has observed that the kinds of changes that occur in the development of one CCS are often repeated in the development of other CCSs, i.e. change is domain-specific. As such, one of the goals of SDS is to acquire an understanding of these 'patterns' of change. It is envisaged that achieving this will facilitate the development of future CCSs, in particular, helping SDS to develop capabilities in the following areas:

1. *Change anticipation.* Anticipate change (e.g. to expect the introduction of new requirements) in order to plan ahead and address change in a pro-active rather than reactive manner.
2. *Change categorisation and change strategy reuse.* Identify and classify different types of change in order to formulate and re-use strategies for coping with change.
3. *Change estimation.* Collect cost and time-scale data for types of change in order to aid estimation (e.g. time and cost to implement a change).

These capabilities, however, depend on the capture and reuse of *domain-specific change knowledge*, which is an aspect of change that is not addressed by the current literature.

3. Current Methods and Techniques for Handling Change

There are a number of distinct research areas that provide an angle for addressing aspects of software evolution, some of which are particularly relevant to change at the requirements level.

- *Process models* for software evolution attempt to define high-level process models that attempt to either reduce or eradicate the perceived gap between software development and maintenance. The IEEE software maintenance model, described by [3], proposes a seven-step 'waterfall-like' process: a) problem identification, b) analysis, c) design, d) implementation, e) system test, f) acceptance test and g) delivery. This, however, provides no real detailed guidance for the software maintainer. The FEAST (Feedback and Software Technology) model [16] is an advancement of Lehman's well-known E-process model. In the FEAST model, the software process is modelled in a dynamic fashion as a continuous system varying over time. Execution of the process is through control loops, which produce 'feedback' to drive the next steps in the process. However, FEAST is a theoretical (rather than practical) model, and questions about the

level of modelling (fine-grain modelling is likely to result in extremely complicated models) and nature of control loops (open or closed) are unclear.

- *Heuristic support* includes guidelines for managing change, which might be appropriate at different levels of support, e.g. managerial-level, project-level, and task-level. [22] identify 3 strategies for change: a) identifying change early in the lifecycle, b) facilitating the incorporation of change and c) reducing change. They offer further guidelines about techniques appropriate to each strategy, e.g. the use of prototyping, simulation, work-throughs and scenarios in strategy a).
- *Evolutionary delivery methods* [7] encourage incremental development and early feedback. Evolutionary delivery methods do, however, rely on being able to compartmentalise the system and to deliver it in stages. The application of evolutionary delivery methods has generally been used at a macro-level to deal with change from a user-perspective, such as in the form of prototyping. However, support for other aspects of change such as studying the impact of change or change anticipation is not addressed.
- *Logic languages* can help to reason formally about change and impact [24, 4]. [4] describes a language with goal-structures that capture the relationships between goals and sub-goals, which can be used to reason about alternative design decisions and analyse trade-offs. One problem here is that there are likely to be many influences on the software process, and a language that attempts to capture all these concepts is likely to be both large and complex (even before one attempts to apply it). In addition, there are validation issues as the ability to do any useful reasoning relies on having realistic and representative models of the real-life process.
- *Taceability* attempts to define the important associative relations between and across actors and artefacts in the software process [20. Traceability is important in determining how change to one software artefact might affect another artefact. Traceability at the implementation level is supported by techniques such as program slicing [24] and program dependence graphs [19]. However, there is an increasing need to extend traceability to earlier levels of software engineering. One issue is the granularity at which traceability is performed. For example, in requirements engineering, traceability might be performed at the requirements document level and/or at the level of individual requirements.
- *Environments* have been proposed to support change impact and change propagation. Environments tend to support a mixture of automatic and user-assisted change operations. Such environments operate at the implementation level and use program dependency graphs [8, 11]. An expert system environment has also been suggested by [1].

The work so far in this area concentrates on general and non-domain-specific methods and techniques for supporting change. Our work differs, in that we are concerned with *domain-specific* methods for supporting change in the software process. It is generally recognised that domain knowledge is central to the enactment of many software processes, e.g. requirements engineering [6]. Our general hypothesis here is that individual application domains exhibit "patterns" of change (particularly at the requirements level), and that understanding these change patterns can facilitate the evolution of future systems in the domain, as well as provide guidance for the application of non-domain-specific techniques. In short, we are assuming that change in

the past (historical change) will be indicative of change in the future. This is not an unusual assumption in software engineering, e.g. historical data forms the basis for reliable estimation models [21].

In the following, we present a view of the change process, called the *change cycle*, which we have developed in our current work. First, however, we feel it is beneficial to discuss what the general, desirable characteristics of a change process are.

4. Desirable Characteristics of a Change Process

A *prescriptive* process describes the activities necessary to complete a goal and the order in which those activities are to be carried out [25]. [17] describes some of the desirable properties of a prescriptive process to be: convey information readily, have reasons supporting their definition, have formally defined syntax and semantics, comprehensive, describe and integrate different levels of abstraction and is reusable. One approach for coping with change is for organisations to develop prescriptive models of the change process. From our discussion in the previous sections, a prescriptive model of the change process should help in some, if not all, of the following ways:

- *Help in change prevention.* Not all change can or should be prevented. However, it is sensible to prevent or least minimise unnecessary or 'risky' change.
- *Help in impact analysis.* It should be possible to reason, or at least conjecture, about the potential effects of change on software artefacts and the software process.
- *Help in change planning.* Change should be planned into the software process rather than being treated as an adjunct to it. Also, the sequencing of change and the collective effect of change over change in isolation should be considered.
- *Help in stability assurance.* After a change has been implemented, assurance procedures should exist to ensure that integrity (software and documentation) is maintained. This is particularly important in the case of safety-critical systems.

Ultimately, dealing properly with change at the time will save on re-work further down the software time-line, just as dealing with problems at the requirements level is more cost-effective than dealing with the problem at implementation.

5. The Change Cycle: A Framework for Change Analysis

A framework for change analysis that we have developed is the change cycle (Figure 1). The change cycle provides a concise and integrated view of the change process that emphasises the *domain-specific* and *reuse* perspectives that are important in our work.

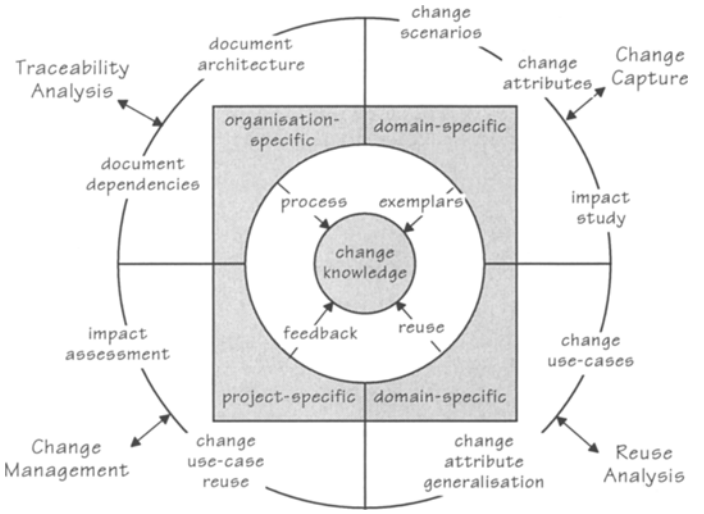

Figure 2 The change cycle

At the centre of the change cycle is an evolving body of *change knowledge*. The change cycle is composed of four distinct sectors which contribute to the change knowledge:

1. *Traceability analysis.* An analysis of the dependencies between and across software artefacts and actors in the software process.
2. *Change capture.* The modelling and capture of change on specific projects.
3. *Reuse analysis.* The derivation of generic change patterns and reusable change knowledge.
4. *Change management.* The application of generic change patterns and reusable change knowledge in the management of change on a specific project.

Steps 2-4 share a similar overall process to that of domain-specific reuse approaches, for example, the reuse of domain-specific requirements as described by [12]. We argue that this should not be surprising, as requirements engineering, like change analysis and management, is a process that is strongly guided by domain expertise [6, 5].

The change process in Figure 2 is a cycle because each iteration of the change cycle refines and re-uses the body of change knowledge. We view change knowledge as something which evolves over time and with development experience in the domain. It is both unreasonable and unrealistic to expect a single analysis of change to miricuously produce a complete and definitive knowledge about change, especially in complex real-world domains (as opposed to academically-bounded domains often used in the literature). Each iteration of the change cycle, however, can contribute to an explicitly documented body of change knowledge.

One role of the process cycle is to organise specific change analysis and management techniques within the broader change process. Table 1 outlines some of the specific techniques which can be used in each sector of the change cycle. The

'Focus of techniques' field in Table 1 describes the general purpose of the set of techniques; the 'Level/scope of usage' indicates the intended scope of these techniques. Our list of techniques is not complete, and refer to the ones which we have used so far in our work. [22] also describe a more general set of techniques which could also be fitted into the context of the change cycle).

Table 1 Techniques in the change cycle

	Focus of techniques	*Techniques*	*Level/scope of usage*
Traceability analysis	Model associative relationships in the existing software process.	Documentation architecture definition Documentation dependency templates	Organisation-specific
Change capture	Acquisition of change knowledge through the analysis of exemplars.	Change scenarios Change attributes Change metrics	Domain-specific
Reuse analysis	Identification and formalisation of reusable change knowledge.	Change use-cases Change attribute generalisation	Domain-specific
Change management	Application of reusable change knowledge and the provision of feedback for its improvement.	Change recognition Change use-case reuse Impact analysis	Project-specific

The grounding for many of these techniques should be familiar to those with an appreciation of basic software engineering methods, e.g. use-case modelling, metrics and traceability. We believe that a systematic process for managing change can be achieved by combining and applying these basic software engineering concepts. We provide an illustration of the use of some of these techniques at SDS in the next section.

6. Application at SDS

Because the CCS domain is real-world domain that is subject to real change and real customers, we feel the domain is a suitable domain upon which to concentrate our research efforts on. In addition, the domain is relatively intuitive (compared to, for example, the author's previous work on aircraft control systems [12]), so the learning curve for understanding the domain did not adversely impede our research. We have currently performed one iteration (which took about one person-week of effort) of the change cycle to study change in the customer complaints domain at SDS. The result of this iteration is an initial body of change knowledge, which we will refine in future iterations. We intend to perform one iteration per CCS developed at SDS, in order to refine our change knowledge at every opportunity. In the following, we describe the application of specific techniques in each sector of the change cycle. However, instead of burdening the reader with the fine details of our application, we draw out the main goal-task activities in each sector.

6.1 Sector 1: Traceability analysis

Goal: Study documentation traceability
Task: Draw the documentation architecture

Just as an implementation has an architecture, so do requirements and designs. The importance of the requirements architecture in requirements reuse is described by [15]. The requirements and design architecture is often reflected in the documentation architecture. The documentation architecture describes the internal structure of system documentation and the dependency relationships that exist between documentation units within the same document and across different documents. At SDS, a CCS requirements document is produced from the generic CCS. From this, a CCS design document is produced. Figure 3 shows the documentation architecture. Here, the small boxes depict the documentation units, and the arrows the dependency relationships between units that we have uniquely identified.

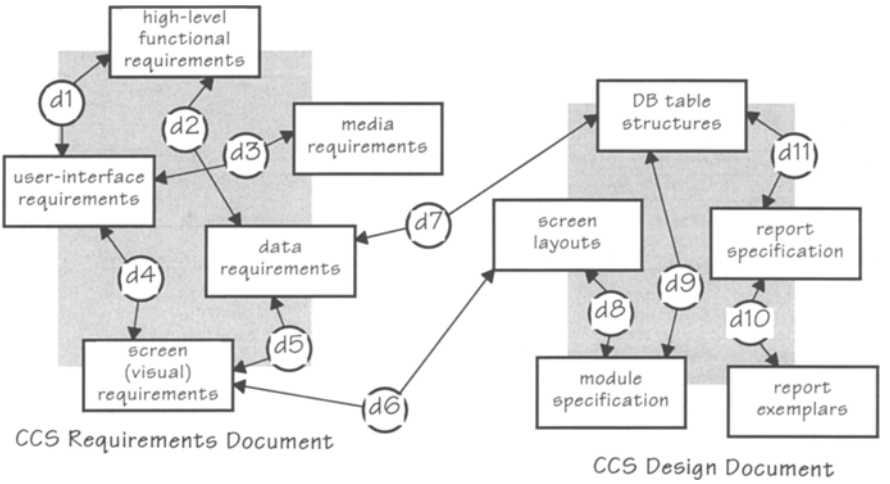

Figure 3 The documentation architecture

Goal: Examine the nature of documentation dependencies
Task: Instantiate documentation dependency templates

We used documentation dependency templates (DDTs) to help examine the nature of dependencies in the documentation architecture, i.e. make clear *why* there is a dependency not just the fact that there is. An example of an instantiated DDT is shown in Table 2.

Table 2 An instantiated DDT

Documentation dependency	D7
Documentation structural unit 1	Data-requirements-CCS-requirements-document
Documentation structural unit 2	DB-table-structures-CCS-design-document
Dependency	Requirements to DB-table-structure.
Consistency rules	1. Data requirements must match the underlying database table structure. 2. Data requirements must be able to be met through processing from data stored according to the underlying database table structure.

We instantiated DDTs for each dependency identified in the document architecture to understand, at a fine-grain level, the traceability concerns in CCS development at SDS.

6.2 Sector 2: Change capture

Goal: Elicit the change process followed by developers for dealing with specific kinds of change
Task: Walk-through change scenarios

An important element of change capture was to understand how staff as SDS deal with different kinds of change. We elicited processes for dealing with change by identifying and walking-through different *change scenarios*. We identified common change scenarios based on an examination of the kinds of changes that had occurred in previous CCSs. We used event traces to represent and discuss change scenarios because of their ease of understanding and semi-rigorous approach. Figure 4 shows an example of a change scenario in the case of a change to a screen layout (i.e. visual requirement).

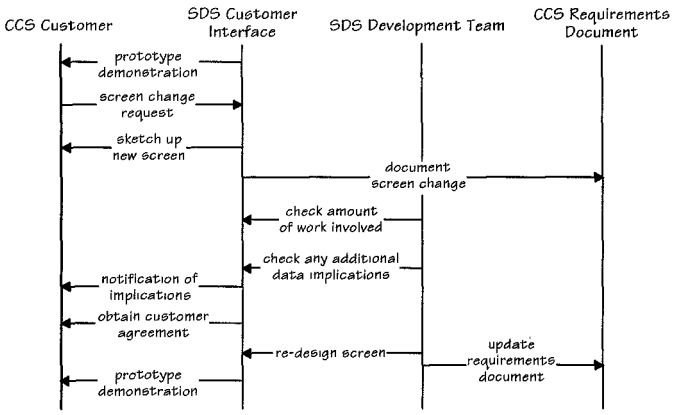

Figure 4 A change scenario

Goal: Model and capture information about specific changes
Task: Characterise the change attributes of a change

Change scenarios capture the process of change. To model other features of change, we have identified a set of change attributes, which we formulated during discussion with staff at SDS. We have classified the change attributes into four categories that reflect the aspects of change of most concern to SDS.

1. *Source.* Where the change emanates from.
2. *Severity.* The severity of the change.
3. *Effect.* The effect that the change has on software artefacts and the software process.
4. *Strategy.* The strategy (such as process and pitfalls) used to deal with the change on a specific project.

Table 3 shows the change attributes (with examples) we have identified so far, but which we acknowledge may not be complete.

Table 3 Change attributes

Attribute Category	*Attribute*	*Example Attribute Values*
What	Description	Add 'counter' display to the new complaints form.
Source	Who (made the change request)	John Smith
	When	Prototype demonstration, 12-10-96
	Why	Counter needed to display number of unresolved complaints
Severity	Time	5 person-hours
	Cost (internal)	£125 at £25/person-hour
Effect	Change artefacts	1. Screen (visual) requirements (CCS Requirements document) 2. Data requirements (CCS Requirements document) 3. Screen layouts (CCS Design document) 4. New module specification (CCS Design document) 5. New complaints form (code) 6. New function in function library (code)
Strategy	Process	1. Re-design new screen layout. 2. Design counter function. 3. Implement new screen. 4. Implement new function. 5. Test
	Pitfalls	1. Adding a new field may require a significant redesign to the original form. 2. Adding a new field may violate screen design consistencies. 3. New field of this type (counter) should 'match' existing fields of the same type. 4. Counter fields should not add or modify data in the underlying database.

Finding the values for change attributes for a particular change takes place over the life of a change, i.e. from the inception of the change through to its completion. For example, the severity of a change in terms of its time and cost can only be known once the change has been completed (though we might have some idea beforehand of its likely severity — this is an example of the kind of reusable knowledge we are trying to exploit).

It should be recognised that what we are doing here is taking a direct modelling approach, i.e. we view change as an explicit concept in the software process just as we view each requirement or each module of code as an explicit concept. Each change in the development of a CCS can be captured in terms of our change attributes. This provides SDS with a convenient way of documenting change, but also enables SDS to build up an empirical and historical record of change upon which to generalise (sector 4 in our change cycle).

6.3 Sector 3: Reuse analysis

Goal: Establish a set of reusable change patterns
Task: Build-up a change use-case hierarchy

Through discussions at SDS over many concrete change scenarios, we were able to identify common change scenarios which we call change use-cases (as in the use-cases described by [10]). Change use-cases are reusable across CCS projects. Change scenarios are instances of a change use-case. For example, our discussions at SDS quickly revealed three common change use-cases:

1. *Visual-change-use-case.* Changes to the screen layout (visual requirements), which are often raised during prototype demonstrations with the customer during an iterative development process.
2. *Report-change-use-case.* Change to reports, especially when 'example' reports are produced from test data and are given to the customer for inspection.
3. *Data-change-use-case.* Change to data requirements, which often includes the inclusion of new information to be recorded by the system (customers tend not be know at the start what their exact data requirements are).

We suspect that there are many more change use-cases that we have not yet identified. In particular, there are also specialised forms of the change use cases mentioned already. A report change use-case, for example, may have a report-layout-change-use-case and a new-data-change-use-case. Building up a complete picture of these change use-cases in the customer complaints domain in terms of a change use-case hierarchy (Figure 5), is part of our on-going work.

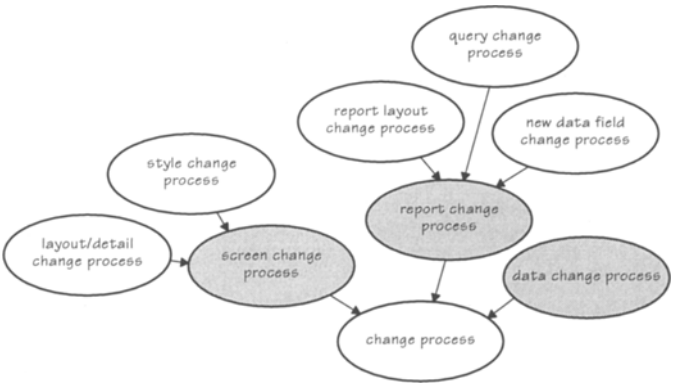

Figure 5 Evolving a change use-case hierarchy

We develop a generic event trace for each change use-case by abstracting, using manual inspection methods, from concrete event traces. The change use-case thus captures reusable process knowledge for dealing with a particular type of change.

Goal: Create reusable change knowledge
Task: Abstract from change attributes

The analysis of many similar change exemplars (similar in terms of being instances of the same change use-case) allows us to define general or typical values for change attributes. The change attributes we have initially focussed on are those in the severity category i.e. time and cost of change (e.g. a data-change will typically take between 15-20 man-hours to complete). It should be noted that productivity models are based on historical data [21]. We are essentially doing a similar thing, but at an earlier stage in the metrics collection process. To give an example, customers often ask for changes to a CCS after it has been installed. One problem for managers at SDS is estimating what the time and cost of the change will be. Presently, SDS relies on management expertise to carry out estimation. However, we can now facilitate this estimation process by providing typical time and cost figures based on past change exemplars. We expect the reliability of these figures to improve over time as the sample size for the number of changes increases.

Goal: Study the impact of change to software documentation
Task: Extend traceability in change scenarios

The change scenarios described earlier help to elicit the change process carried out by staff at SDS. We have found that change scenarios are also a good way of validating change processes against the documentation architecture. For example, when a developer at SDS says 'update requirements document' in a change scenario, we are able to question what part of the requirements document, i.e. the documentation unit, and then use documentation dependencies to ascertain what checks need to be done on related parts of the documentation. If we had not done this, changes addressed and documented 'locally' but not in other dependent parts of the document architecture may leave the documentation in an inconsistent state. From this we are able to formulate a number of *change heuristics* of the form:

WHEN change IS A screen-change
THEN UPDATE screen-requirements IN CCS-requirements-document
AND REQUIREMENTS-CHECK user-interface-requirements IN CCS-requirements-document
AND REQUIREMENTS-CHECK high-level-functional-requirements IN CCS-requirements-document
AND DESIGN-CHECK screen-layouts IN CCS-design-document...

Change heuristics formalise the 'what' to change, 'what' to check and 'when' to check. We see change heuristics as essential in system maintenance situations where the original CCS developer is not the same person who carries out the maintenance. We also believe that change heuristics (though not necessarily of the same form as here) are central to providing knowledge-based support for the change process.

6.4 Sector 4: Change management

Goal: Manage change on a specific project
Task: Tailor the generic change management process

Change management is the fourth sector of the change cycle and is concerned with the management of change on a specific project. The general management process is outlined in Figure 6, which begins with a change request from the customer.

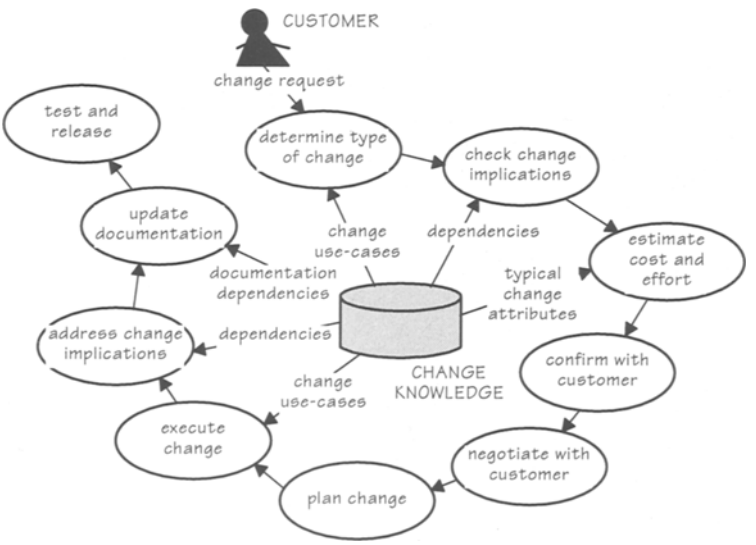

Figure 6 General change management process

At the centre of the change management process is the body of change knowledge which is at the centre of the change cycle. We propose that projects tailor the change management process for specific projects in order to define a standard change man-

agement process. Such processes can be used to provide practical guidance for dealing with change, especially when encoded within a process-driven software-engineering environment.

Goal: Resolve viewpoint problems of change
Task: Issue and use a 'Change processes' document

One of the problems that we have come across is that change is viewed differently from different viewpoints. For example, the 'salesperson' that interacts with the customer might consider a change to be relatively 'easy'. However, the same change might be considered extremely difficult from a developer viewpoint, e.g. because the developer has made some (not unreasonable) assumption. In this kind of case, there is a mismatch between the individual perceptions of change. Resolving viewpoint problems requires working towards a common understanding of change processes between members of a project. One approach is to produce and use a 'change processes' document detailing the kind of change knowledge that we have described in this paper (change use-cases, change attributes etc.). Such a document could be used as a reference document in different scenarios, e.g. change estimation and customer negotiation.

7. Tool Support for Change

Captured change knowledge can be exploited in automated or semi-automated tool support for change. We are currently investigating the architectural requirements for tool support for change in SDS. A proposed 'high-level' architecture for a change environment, based on our work with the change cycle at SDS, is shown in Figure 7. There are two types of users, one type who is concerned with the input of change knowledge (post project perhaps), and the other type who is looking for help during a current project.

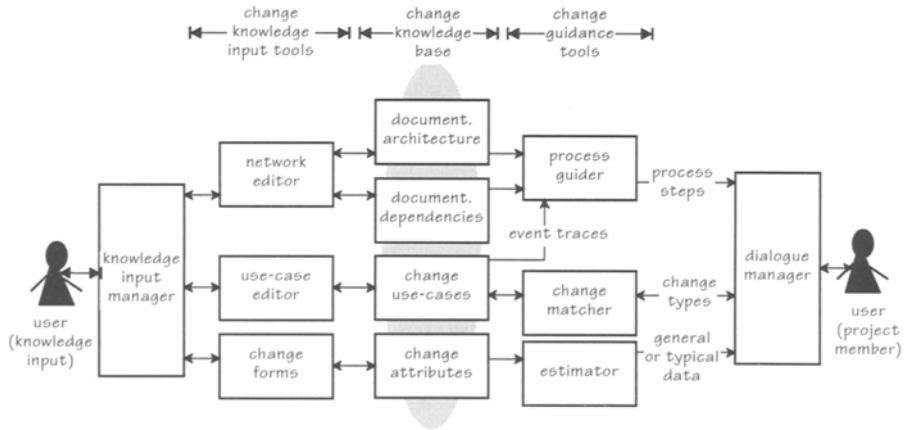

Figure 7 Proposed high-level architecture for a change environment

The change-environment is comprised of three kinds of tools. First, change knowledge input tools which allow the input of change data and knowledge, such as a network editor for 'drawing' out a document architecture. Second, a change knowledge base (KB) for storing change knowledge (though we have not yet worked out the exact representation for the KB). Third, change guidance tools that are able to process change knowledge and provide guidance on a specific project. We see three main change-guidance tools: a change matcher, process guider and estimator. The change matcher matches helps the user match the current change to a change use-case in the change KB. The process guider assists the user by proposing steps for dealing with the change. These process steps are derived from the event trace in the change use-case and by tracing documentation dependencies in the document architecture. The estimator is a tool that is able to generalise from a large collection of change instances. For example, one function of the estimator would be to produce the average time it takes to complete a particular kind of change.

8. Evaluation Approach

Our evaluation approach is loosely based around a Goal-Question-Metric [2] paradigm, and on two kinds of cycle which 'track' the process cycle (Figure 8).

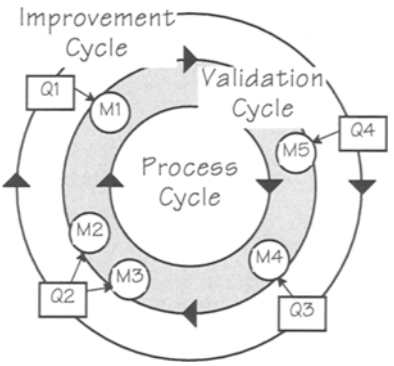

Figure 8 Improvement and validation cycles

The two kinds of cycle are:

1. *Improvement cycle.* This is concerned with the establishment of improvement goals. Typical goals that we have found relevant include (G1) reduction in the overall level of change, (G2) reduction in the level of change for a particular type of change, (G3) faster turnaround in completing change requests from the customer.
2. *Validation cycle.* This is concerned with the establishment of metrics or indicators (because it might be difficult to metricate some aspect of an improvement goal) that supports the improvement goals. Metrics and indicators include (M1) total number of changes (supports G1), (M2) total number of changes pertaining

to a particular change use-case category (supports G2) and (M3) average effort taken to complete a change (supports G3).

At the time of writing, we have just initiated the collection of metric data. However, we have found that establishing appropriate metrics for change is in itself a non-trivial problem for which there is little discussion of in the current literature. The identification of appropriate and meaningful (to SDS) metrics is part of our on-going work. Our initial experiences suggest that insightful metrics on change is unlikely to be supported without established change procedures in place within the organisation, as exhibited by detailed change request forms and change monitoring tools (as in the change environment proposed earlier). Essentially, we can not do any detailed reasoning on change if the change data is not there in an amenable form. Part of our strategy here has been to choose change attributes that reflect the kind of analysis that we wish to perform.

9. Conclusions

This paper has described efforts at change analysis and management in a commercial setting. The starting point for our work was the fact that the systems being developed all pertained to the same domain (namely, the CCS domain). This has encouraged us to take a *domain-specific* approach to change analysis and management, which differs in emphasis from the existing work on change we reviewed in Section 3. Our main contribution to this area is the 'change cycle', which attempts to provide a concise and integrated view of the change process. We have shown how the change cycle has been applied to address aspects of the change problems in the CCS domain at SDS. One area that we feel we need to elaborate more on in our work is the relationship between the change process and perceived models of the software process, such as Waterfall or Prototyping. This is part of our on-going work to develop a 'change-enhanced' version of a prototying-centred software development approach. As we noted in Sections 7 and 8, tool support for the change process and change metrics are two further areas of on-going work which we hope to report on in more detail in future papers.

10. References

1. Avellis, G. (1992), CASE support for software evolution: A dependency approach to control the change process. In proceedings of 5th International Workshop on Computer-aided software engineering, pp.62-73, Montreal, Canada, July 1992.
2. Basili, V. and Weiss, D. (1984), A method for collecting valid software engineering data, IEEE Transactions on Software Engineering, November 1984, pp.728-735.
3. Bennett, K. (1996), Software evolution: past, present and future, Information and software technology, 38:673-680, 1996.

4. Chung, L., Nixon, B. and Yu, E. (1997), Dealing with change: an approach using non-functional requirements, Journal of Requirements Engineering, 1(4), 1997.
5. Curtis, B., Kellner, M.I. and Over, J. (1992), Process modelling, Communications of the ACM, 35(9), 1992.
6. Fickas, S. and Nagarajan, P. (1988), Critiquing software specifications, IEEE Software, November, 1988.
7. Gilb, T. (1988), Principles of Software Engineering Management, Addison-Wesley, England.
8. Han, J. (1997), Supporting impact analysis and change propagation in software engineering environments, In Proceedings of the 8th IEEE International Workshop on Software Technology and Engineering Practice, London, UK, 14-18 July, 1997.
9. Harker, S., Eason, K. and Dobson, J. (1993), The change and evolution of requirements as a challenge to the practice of software engineering, In proceedings of the IEEE international symposium on requirements engineering (RE'93), San Diego, California, 1993.
10. Jacobson, I., Griss, M and Jonsson, P. (1997), Software reuse: architecture, process and organisation for business success, ACM Press, New York.
11. Kaiser, G., Feiler, P. and Popovic, S. (1988), Intelligent assistance for software development and maintenance, IEEE Software, 5(3):40-49, May 1988.
12. Lam, W. (1997), Achieving Requirements Reuse: a Domain-Specific Approach from Avionics, Journal of Systems and Software. 38(3): 197-209, 1997
13. Lam, W., McDermid, J.A. and Vickers, A.J. (1997), Ten Steps Towards Systematic Requirements Reuse (expanded version), Journal of Requirements Engineering, 2:102-113, 1997.
14. Lam (1998a), Managing requirements evolution and change, IFIP WG2.4 Working conference: Systems implementation 2000: languages, methods and tools, Berlin, Germany, February 23-26, 1998.
15. Lam, W. (1998b), A Case-study of Requirements Reuse through Product Families, Annals of Software Engineering (to appear).
16. Lehman, M. (1996), Feedback in the software evolution process, Information and Software technology, 38:681-686, 1996.
17. Madhavji, N. (1991), The Process Cycle, Software Engineering Journal, September, 1991.
18. Madhavji, N. (1997), Panel session: impact of environmental evolution on requirements changes, In Proceedings of the 3rd IEEE International Conference on Requirements Engineering, 1997.
19. Podgurski, A. and Clarke, L. (1990), A formal model of program dependencies and its implication for software testing, debugging and maintenance, IEEE Transactions on Software Engineering, SE-10(4):352-357, 1984.
20. Pohl, K., Domges, R. and Jarke, M. (1997), Towards method-driven trace capture, 9th International Conference, CaiSE'97, Barcelona, Spain, June 1997.
21. Putman, L. and Myers, W. (1992), Measures for excellence, Reliable software on time, within budget, Yourdon Press, Prentice-Hall, New Jersey, 1992.
22. Sugden, R. and Strens, M. (1996), Strategies, tactics and methods for handling change, In Proceedings of International IEEE Symposium and Workshop on Engineering of Computer-Based Systems (ECBS '96), Friedrichshafen, Germany, March 11-15.

23. Yu, E. (1997), Towards modelling and reasoning support for early-phase requirements engineering, In Proceedings of the 3rd IEEE International Conference on Requirements Engineering, 1997.
24. Weiser, M. (1984), Program slicing, IEEE Transactions on Software Engineering, SE-10(4):352-357, 1984.
25. Zave, P. (1986), Let's put more emphasis on prescriptive methods, ACM Sigsoft Software Engineering Notes, 11(4):98-100.