ISSN 1744-1986

Technical Report N° 2008/ 11

Introducing new features to a critical software system

Thein Than Tun
Rod Chapman
Charles Haley
Robin Laney
Bashar Nuseibeh

3rd July, 2008

Department of Computing
Faculty of Mathematics, Computing and Technology
The Open University

Walton Hall, Milton Keynes, MK7 6AA
United Kingdom

http://computing.open.ac.uk

Introducing new features to a critical software system

Thein Than Tun[†] Rod Chapman[‡] Charles Haley[†] Robin Laney[†] Bashar Nuseibeh[†]

[†]Department of Computing
The Open University
Walton Hall, Milton Keynes, UK
{t.t.tun, c.b.haley, r.c.laney, b.nuseibeh}@open.ac.uk

[‡]Praxis High Integrity Systems Limited
20 Manvers Street
Bath, UK
rod.chapman@praxis-his.com

Abstract

In response to changing requirements and other environmental influences, software systems are increasingly developed incrementally. Successful implementation of new features in existing software is often difficult, whilst many software systems simply 'break' when features are introduced. Size and complexity of modern software, poor software design, and lack of appropriate tools are some of the factors that often confound the issue. In this paper, we report on a successful industrial experience of evolving a feature-rich program analysis tool for dependable software systems. The experience highlights the need for a development framework to maintain rich traceability between development artifacts, and to satisfy certain minimal necessary conditions of artifacts during and after the implementation of a new feature.

1 Introduction

Software development is increasingly concerned, not with creating software from scratch, but with modifying and extending existing software to satisfy new requirements [25]. Incremental development of new features in feature-rich software, which we call *feature-based development*, represents a substantial and interesting class of development that we investigate in this paper. Intuitively, a software feature represents a recognizable unit of functionality that satisfies some user requirements. Since the introduction of new features often 'breaks' functioning software, feature-based development signifies an important challenge that threatens software quality.

With its emphasis on reuse, characterization of requirements as features, and notion of continual development of software, feature-based development has much in common with component-based development [29, 9], product-line engineering [6, 2] and software evolution approaches [21, 24, 10, 5].

However, our notion of feature-based development differentiates itself in the following ways: (i) a current system already exists and customers of the system wish to make it also satisfy new requirements, (ii) the current system does not belong to a product family, (iii) features do not necessarily mean variability in architecture, and (iv) the main concerns relate not only to the evolution of code but also to the evolution of problem structures and requirements. Therefore, we argue that this is a unique development scenario that deserves to be studied in its own right.

In this paper we report on an industrial experience of evolving a feature-rich software tool used in the development of dependable systems. In particular, this report highlights the necessity for maintaining the relationships between evolving development artifacts, and the minimal necessary conditions to satisfy during and after implementation of new features. The benefits of deploying such a framework in practice include: (i) improved software quality due to rich traceability between (evolving) artifacts, and (ii) an assurance that the software will continue to work after new features have been introduced.

The paper is organized as follows. Section 2 sets the discussions into context by surveying related work. Section 3 explains what is meant by problem structure and why it is important according to a conceptual view of software development. Based on that conceptual view, we describe a framework for feature-based development in Section 4, which is used to describe the industrial experience from an ongoing development of a program analysis tool for high integrity software in Section 5. Section 6 highlights the lessons learnt and provides concluding remarks.

2 Related Work

Turner et al [30] suggest a conceptual basis for feature engineering of software systems, and argue that by organizing requirements and life-cycle artifacts along their proposed notion of features, the gap between the problem space and solution space can be bridged. Although we have similar motivation, we believe that a more formal framework to reason about traceability between these artifacts helps improve software quality [15]. Jackson [19] formalizes the relationships between major artifacts in software development, which is further developed by Gunter et al [11] and later Hall and Rapanotti [13] to provide specific proof obligations for (or responsibility of) stakeholder groups. The conceptual framework for feature-based development proposed in this paper is based on the original work by Jackson [19].

Acknowledging that the term feature has specific meanings in certain areas of research such as product-line engineering, and feature interactions, we adopt the general notation of feature as a unit of system functionality that is "user accessible" [17]. Schobbens et al [28] survey feature diagrams and propose a formal semantics of their diagrams. Chen et al [8] show how feature analysis can help address cross-cutting concerns in architecture, whilst Reiser and Weber [27] discuss the limitations of traditional feature trees in describing large and complex systems, and propose an approach to overcome various limitations. There is a considerable literature on feature interaction problems in telecommunication [7] and other application domains such as email [14]. Hay and Atlee [16] treat feature interactions as a more general software problem and discuss a feature composition approach to resolve them.

3 Problem Structures and Software Development

Requirements of a software-intensive system are rooted in its environment, which has some identifiable entities that are related to each other. Understanding what the software needs to do requires knowledge of these entities and their relationships with the system, which we call a *problem structure*. Consider the sea buoy control system in Fig. 1, whose requirement, written inside the dotted oval in the diagram, is to periodically broadcast information about sea water temperature. In order to write a specification for the controller, one needs to consider its problem structure. This usually involves identifying relevant entities in the environment and their properties, and asking such questions as: How does the temperature of sea water affect sensor readings? How and when does the sensor inform the controller of its reading? What does the controller need to 'say' to the radio transmitter to broadcast temperature information? If one does not know the problem structure in this case, one cannot write a specification for the controller.

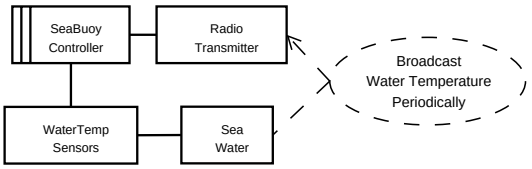


Figure 1. A Simple Problem Structure of a Sea Buoy Controller

This example highlights the need for understanding relationships between software development artifacts. Jackson [19] identifies five artifacts in system development – domain knowledge (W), requirements (R), specifications (S), programs (P) and the programming platform or computer (C) – and describes their general relationships using the logical entailment operator ($\vdash$) as follows.

$$W, S \vdash R$$
$$C, P \vdash S$$

The first entailment ($W, S \vdash R$) differentiates between specifications and requirements by suggesting that specifications, *within a particular physical (world) context*, imply requirements. In other words, specifications rely on explicit domain properties in satisfying the requirements. Similarly, the second entailment ($C, P \vdash S$) differentiates between programs and specifications by suggesting that programs, *on a particular programming platform*, imply specifications. Programs, therefore, rely on properties of the programming platform in satisfying the specifications. We view the strength of the logical entailment operator in these formulae to be non-prescriptive, meaning that (i) the artifacts (W, R, S, P and C) may be described in varying degrees of formality and (ii) showing that an entailment relationship holds for some given artifacts does not necessarily require mathematical proofs: informal arguments are often sufficient.

In this sense, the two entailment relationships provide a general framework for establishing and maintaining traceability links from requirements to program code, by factoring out properties of the world and the programming platform. Additionally, the entailment relationships help define responsibilities of various stakeholders. In broad terms, the first entailment is the responsibility of requirements engineers, and the second entailment, that of developers.

Finally, problem structures of software to be developed from scratch have different characteristics from those of software to be developed incrementally by modifying and extending an existing system. In the latter case, appropriate representation of the existing program as a partial solution to the future problem poses an important question. The next

section discusses how this question can be put into the context of the framework above.

4 Feature-based Development

In a typical feature-based development project, there is an existing solution that satisfy current requirements. In particular, there is a problem R_{now} in the present state of the world W_{now}, and a specification of the current machine, S_{now}, to solve the problem such that:

$$S_{now}, W_{now} \vdash R_{now} \tag{1}$$

The current program P_{now}, implemented on a particular computer, C_{now}, satisfies the specification S_{now}:

$$P_{now}, C_{now} \vdash S_{now} \tag{2}$$

Customers of this system want a new system in future, so that:

$$S_{future}, W_{future} \vdash R_{future} \tag{3}$$

and the new system continues to satisfy requirements for the existing system:

$$S_{future}, W_{future} \vdash R_{now} \tag{4}$$

This entailment relationship (4) captures an important property of systems in feature-based development considered in this paper. As discussed in the next section, it serves as a basis for an important proof obligation in feature-based development.

Customers need a new program, either on the same or a different computer – we restrict ourselves to the former in this work – which satisfies the future requirements as specified in S_{future}:

$$P_{future}, C_{now} \vdash S_{future} \tag{5}$$

Importantly, developers do not wish to develop the system from scratch – that is to say, refine R_{future} to P_{future}. Rather, they wish to reuse P_{now}.

First we discuss how P_{now} can be represented in the problem structure of R_{future}.

4.1 Representing the Current Solution

A key question feature-based development needs to address is that of representing the existing solution. If we take a rather formal view of the development, we may use the following process. First, obtain the new requirements R_{new}, so that R_{now}, $R_{new} \vdash R_{future}$. Since P_{now} is already implemented on C_{now}, describing P_{now} running on C_{now} as some given properties of W_{future} means (i) P_{now} is reused as it is (ii) S_{new} (or specification for R_{new}) has to acknowledge the existence of S_{now} and takes into account potential concerns that may arise from when implementation of S_{new} is composed with P_{now}. For example, there could be shared variables between S_{now} and S_{new}, and implementation of S_{new} must not invalidate assumptions S_{now} has on those shared variables. Taking such concerns into account, refining S_{new} to P_{new} will lead to a program that will compose with P_{now}, producing the required P_{future}.

This view assumes (i) developers do not modify P_{now} and (ii) P_{new} may be delivered in a single increment. Architecture of certain software such as product-line applications may allow these assumptions, but for other systems, these assumptions are not practical. The alternative approach suggested here recognizes that in feature-based development projects, P_{now} is usually modified and P_{new} is rarely built in one increment.

Allowing P_{now} to change offers potential benefits. In related work [23, 26], we have established that there are advantages in letting solution structures influence problem structures. It should be recognized that P_{now} may be a piece of software that has evolved over time, and its current structure may not facilitate eventual composition with P_{new}. Therefore, structural changes to P_{now} to improve its modularity often simplify composition. As well as the benefits, there are potential risks: it is often difficult to understand the full impact of a particular change. The next section discusses a systematic approach to introducing change.

4.2 Introducing New Features: From P_{now} to P_{future}

Major milestones in the implementation of new features are called *releases*, and first we are concerned with the development process between one release and the next. Each release is expected to deliver partial or fully-fledged new features, or elaboration of existing features. Between two consecutive releases, such as P_{now} and P_{future}, there are typically a series of builds implemented and integrated with the current software. These builds can be seen as a series of steps developers take to get from release P_{now} to P_{future}. Assuming that $S_{build-x}$ is a specification for an intermediate build and $W_{build-x}$ is the state of the world (for $P_{build-x}$) between P_{now} and P_{future}, a simple reformulation of the entailment relationships (4) gives an important proof obligation for developers during these steps:

$$W_{build-x}, S_{build-x} \vdash R_{now}. \tag{6}$$

This entailment relationship suggests that, between releases, developers can go about implementing any change they think necessary in the software (and the world) as long as the requirements for the last release are still satisfied.

Note that the entailment relationship (4) cannot hold between releases, and (6) ensures that each accepted build does not break the software, i.e. R_{now} is still satisfied, although the software may have been temporarily broken between two consecutive builds. R_{now} is therefore satisfied during builds between releases (by the entailment relationship 6), and on each release (by the entailment relationship (4)). Setting such a minimal condition gives developers some freedom to explore various design options without breaking the system. How this obligation is honored in practice depends on the nature of software and the implementation technique chosen. (In the next section, we describe one such technique used to achieve this.) The entailment relationships (3) and (5) are other proof obligations when the new new features are implemented.

Often there are a number of possible paths to get to P_{future} from P_{now}, and these builds also allow developers to backtrack, if necessary, to any earlier step in the development. The reason for such caution to change is twofold. From the point of view of users/customers, continued availability of much of existing system behavior in new releases is preferable for a number of reasons including psychological and financial. From the point of view of developers, introducing a small change, the impact of which they understand, is less likely to cause unintended effects on the system behavior.

The next section reports on an experience of introducing a new feature to a critical software.

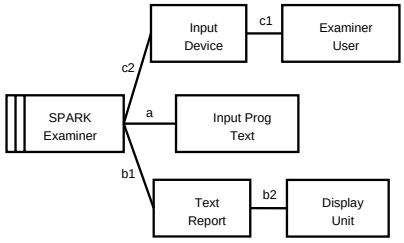
5 SPARK Examiner Case Study

Designed for high integrity software, SPARK is a programming language that uses a subset of standard Ada, with additional annotations written as Ada comments. Ada compilers ignore these comments but SPARK tools use them to do various analyses. SPARK Examiner, or simply Examiner, is one of the tools, which, among other things, checks SPARK program source files for potential run-time errors, such as buffer overflows. The tool, itself written in SPARK, has about 120K lines of code and has been used in development of critical applications for several decades [3, 12, 18].

In the remainder of this section, we describe how a new feature, which is representative of many other features in the software, was added to a particular version of Examiner. Despite its apparent simplicity, implementation of the feature was a major development exercise.

5.1 Current Examiner

The current release of Examiner[1] has a command-line interface, which users use to issue commands to the software. Fig. 2 shows a context diagram of the current Examiner before it was implemented. A context diagram bounds the software problem, and it has two main parts: (i) *problem domains* representing parts of the physical world with certain properties relevant to the problem, and (ii) the *machine domain* or software of the computer the developers must build in order to effect the required properties of the world [20].



a: SE!getProg
c1: EU!keystrokes
b1: SE!makeTextReport
c2: ID!commands
b2: TRdisplayText

Figure 2. Annotated Context Diagram for Current Examiner

Problem domains of the current Examiner, the main parts of W_{now} in this case, are represented by single-lined rectangles in the diagram. The domains Examiner User, Input Device, Input Prog Text, Text Report and Display Unit represent, respectively, user of Examiner software, keyboard, a computer file containing a SPARK program that the user wishes to have the program analyzed, the text of the analysis report for the program (such as error messages and warnings), and the computer display device on which the text report is displayed.

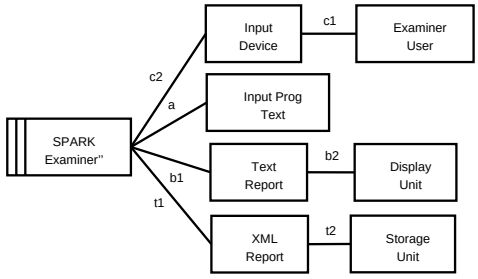
A solid line between domains represents shared phenomena, which are states and properties visible to the domains involved. For example, c1 suggests that there is a property call keystrokes that is visible to both ID (Input Device) and Examiner User (EU), and EU! indicates that the property is controlled by EU, meaning that EU may change the values of keystrokes but ID may not.

The machine domain of Examiner, or S_{now}, is represented by the rectangle with two vertical stripes Fig. 2.

R_{now} of Examiner can be expressed informally as follows:

> *When the user types in commands through the keyboard, Examiner checks the syntax of the command, and if the command is meaningful, Examiner reads in the program text and after appropriate analysis, produce an ASCII report that is displayed in the display unit; and if the command is not meaningful the report contains appropriate*

[1] A reference to a version of Examiner prior to the implementation of the new feature discussed.



a:	SE"!getProg	c2:	ID! commands
b2:	TRdisplayText	c1:	EU! keystrokes
t2:	XP!storeXMLFile	t1:	SE"!makeXMLReport
b1:	SE"!makeTextReport		

Figure 3. Annotated Context Diagram for Future Examiner

error messages.

When the descriptions of the world domains, and specification of the machine in Fig. 2 are produced, they satisfy the above requirements (according to entailment relationship (1)), and when SPARK Examiner (SE) is implemented, the program on the current computer satisfies the specification (according to the entailment relationship (2)). In other words, the existing Examiner satisfies current user requirements.

5.2 Future Examiner

Although current features satisfy many user requirements, a customer had requested that, in order to facilitate data transfer between tools, the same report be produced in a stored file in the XML format. Therefore, the future Examiner needed to satisfy the following requirement:

> *In addition to the existing features, SPARK Examiner should provide a new feature which optionally allows users to save the program analysis results in an XML file (named 'report.xml') on the disk. This XML feature may be provoked by issuing a* `/xml` *switch in the command to SPARK Examiner.*

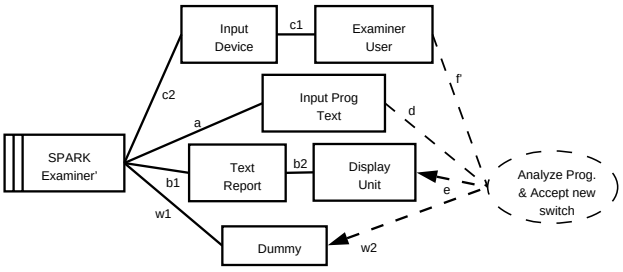
The context diagram for the new Examiner that would include the XML report feature can be described as shown in Fig. 3. The new domains XML Report and textsfStorage Unit in the diagram represent the program analysis result prepared according to a certain XML format, and the disk unit onto which the XML report is to be stored. Notationally, the diagram is unorthodox in one particular sense: with the identity of the new machine SE", we intend to suggest that this is not a new machine to be built from scratch, but an extension of the implemented SE. When the future Examiner is implemented, given the properties of physical domains in future, the new requirement also needs to be satisfied (according to the entailment relationships (3) and (5)).

It is important that the introduction of this new feature does not result in changes in system behavior leading to current requirements not being satisfied. There are a number of reasons for this, including backward compatibility: for example, potential run-time errors recognized by the current tool must also be recognized by the future releases. Furthermore, implementation of this new feature needs to be transparent to segments of customers whom this feature is not provided.

5.3 Introducing the New Feature

The current Examiner in Fig. 2 would reject the new command switch /xml by the user requesting to generate the XML report. In the first build towards implementing the new feature, developers attempted to make the Examiner accept the new switch. The first subproblem can therefore be described informally as:

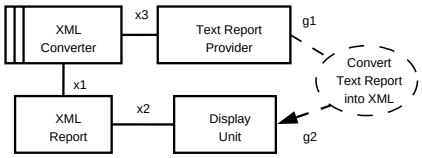
> *When the user issues the new switch, the system, instead of rejecting it, acknowledges the acceptance by causing such a change in the dummy domain, whilst preserving the existing behavior.*



f:	EU!UserCmd	a:	SE!getProg
c2:	ID! commands	d:	IT! ProgText
b2:	TRdisplayText	w1:	SE!DummyText
c1:	EU! keystrokes	w2:	Dummy!DummyText
b1:	SE!makeTextReport	e:	DU!ReportDisplayed

Figure 4. Subproblem 1: Accept the new switch

The reason for choosing to tackle this subproblem first is to define the contract, or interface, between Examiner and the future component implementing the new feature. When implementing this new component, developers will have the knowledge that the new component would recompose with



g2: DU! XMLRep
g1: TP! TextRep
x2: XR!displayXML
x1: XC!makeXMLReport
x3: TP!DummyTextReport

Figure 5. Subproblem 2: Convert Text to XML

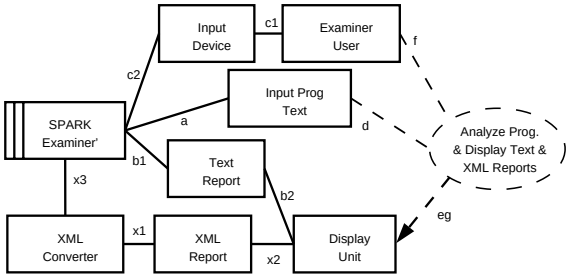
the would-be SE, which is important for them. However, it is emphasized that that this change in structure of SE must not alter the existing behavior in unexpected ways, as suggested by the entailment relationship (6).

Fig. 4 shows the problem diagram for this subproblem. In addition to the two types of domains used in context diagrams, problem diagrams also show the requirement inside a dotted oval. The dotted straight lines denote that when one checks satisfaction of the requirement, one refers to the properties of Examiner User and Input Prog Text – represented by dotted lines without arrowheads, and observes the effects in Display Unit and Dummy – represented by dotted lines with arrowheads. The diagram suggests that when the Examiner User issues the command with the currently accepted switches, the system will display appropriate analysis report for the Input Prog Text; if the new switch is also issued, textual properties of the dummy will also be manipulated in the determined way to acknowledge that the new switch has been accepted. In doing so, developers had created a hook point for the new component, namely w1 in Fig. 4.

To check that the behavior of SE prior to this change has been preserved, developers run the current test cases, make sure that the modified software pass the test cases that the original software passed. This is in accordance with the entailment relationship (6). If developers wanted to gain further confidence, they would release this version of software (which could include other unrelated and visible changes) and evaluate user feedback. When developers are confident that the initial redesign has not resulted in the system behaving in unexpected ways, they proceeded to create a new component to solve the following subproblem:

Convert a text report given by a text report generator into an equivalent XML report.

Fig. 5 shows the problem diagram for this subproblem. In some other cases, such an increment might only contain a smaller subset of the entire requirement for the new feature; for example this XML Converter might generate only essential elements of the XML report, leaving the remaining elements to be added in future build(s)/release(s) after this increment was shown to work. When the XC is shown to be producing the correct XML results, developers will integrate the new component with SE'. Here developers took two smaller steps rather than a large one. Instead of sending the XML report to the storage unit, the first increment displayed but did not store the XML report. Since w1 in Fig. 4 and x3 in Fig. 5 have the same structure, the new component can be plugged in at that point. Fig. 6 shows the problem structure of the third subproblem, which can be described as:



f: EU!UserCmd
a: SE!getProg
d: IT! ProgText
c2: ID! commands
x2: XR!displayXML
b2: TRdisplayText
c1: EU! keystrokes
x1: XC!makeXMLReport
b1: SE!makeTextReport
eg: DU!ReportsDisplayed
x3: SE!DummyTextReport

Figure 6. Subproblem 3: Display text and XML reports

Display the text and XML reports.

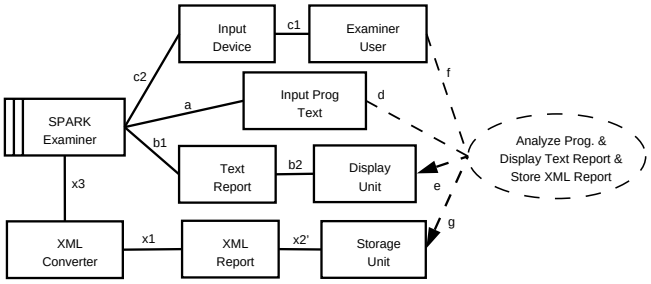
When the XML Converter was producing correctly formatted XML report, developers then reconfigured the domains, and adjusted shared phenomena, so that the text output is displayed and the XML output is stored on the storage unit, thus addressing the following subproblem:

Display text report and store the XML report.

Fig. 7 show the problem structure of the last subproblem, which, when implemented, brings the development very close to the required software, i.e. SE plus XC in Fig. 7 is similar to SE" in Fig. 3. When this increment was proven to work well, developers made the XML report elaborate by adding further details to it in the build. This is to satisfy last entailment relationships (3) and (5).

5.4 Summary

Development practices that we observed in the implementation of the XML and several other features in Examiner are in line with the systematic framework suggested in



a:	SE!getProg	f:	EU!UserCmd
d:	IT! ProgText	c2:	ID! commands
b2:	TRdisplayText	x3:	SE!TextReport
c1:	EU! keystrokes	x1:	XC!makeXMLReport
g:	SU!XMLFileStored	x2:	XR!storeXMLFile
b1:	SE!makeTextReport	e:	DU!ReportsDisplayed

Figure 7. Subproblem 4: Final Increment

Section 4. First, P_{now} was modified, rather than treated as a black-box. In terms of honoring the obligation during the increment (entailment relationship (6)), developers run test cases for P_{now} after each build, which is a kind of regression testing [4]. For each release, a similar test is carried out for the entailment relationship (4). For the entailment relationship (5), developers run new test cases, and for the relationship (3), they relied on further tests and customer feedback.

6 Lessons Learnt and Conclusions

In this paper, we identified a particular type of software development in which existing software needs to be modified and extended to satisfy evolving needs of customers, which we termed feature-based development. Despite some similarities with component-based, product-line engineering and software evolution approaches, feature-based development has distinct characteristics. Based on a conceptual view of software development, we described a framework for introducing new features into a feature-rich software. Using the framework, we described the Praxis experience of developing the SPARK Examiner tool over several years. The main lessons learnt are:

- feature-based development of critical software systems is difficult, but the use of a formal framework for maintaining rich traceability between evolving artifacts provides a good foundation for developing dependable software systems, and
- specific proof obligations for developers during and at the end of each software release are helpful tackling the problem of new features breaking functioning software.

Although our framework was used to describe the development of a critical software tool, it does appear to have wider applicability. For example, the agile and open source development literature [1, 22] refers to practices of nightly builds and continuous integration, which seems to be in agreement with the framework used here. However, further work is necessary to understand how this approach can be put to work in different contexts.

The framework does not consider a development scenario where an existing feature is discontinued in the next release, i.e. where $S_{future}, W_{future} \vdash R_{now}$ does not hold, but not to the extent that P_{now} is completely redundant. The framework also does not characterize formally features and builds. We intend to explore these issues in our future work.

7 Acknowledgements

We are grateful for the support of our colleagues in the Department of Computing at The Open University, in particular, Jon Hall, Lucia Rapanotti and Michael Jackson. We also thank the late Peter Amey of Praxis High Integrity Systems for providing us with help in selecting an appropriate project to study, for access to the project documentation and staff, and for hosting our visit to Praxis. Finally, thanks to the EPSRC, UK for their financial support.

References

[1] S. Augustine, B. Payne, F. Sencindiver, and S. Woodcock. Agile project management: Steering from the edges. *Commun. ACM*, 48(12):85–89, 2005.

[2] F. Bachmann and L. Bass. Managing variability in software architectures. In *SSR'01*, page 126132, Toronto, Ontario, Canada, 2001. ACM Press.

[3] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[4] B. Beizer. *Software Test Techniques (Second Edition)*. International Thomson Computer Press, 1990.

[5] K. H. Bennett and V. T. Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering (ICSE'00)*, pages 73–87, New York, NY, USA, 2000. ACM Press.

[6] J. Bosch. *Design & Use of Software Architectures - Adopting and Evolving a Product-line Approach*. Addison-Wesley, Great Britain, 2000.

[7] E. Cameron and H. Velthuijsen. Feature interactions in telecommunications systems. *IEEE Communications Magazine*, 31(8):18–23, 1993.

[8] K. Chen, H. Zhao, W. Zhang, and H. Mei. Identification of crosscutting requirements based on feature dependency analysis. In *IEEE International Conference on Requirements Engineering (RE'06)*, pages 300–303, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[9] D. D'Souza and A. Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, Reading, MA, 1999.

[10] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Trans. Softw. Eng.*, 27(1):1–12, 2001.

[11] C. A. Gunter, E. L. Gunter, M. Jackson, and Z. Pamela. A reference model for requirements and specifications. *IEEE Software*, 17(3):37–43, 2000.

[12] A. Hall and R. Chapman. Correctness by construction: Developing a commercial secure system. *IEEE Software*, 19(1):18–25, 2002.

[13] J. Hall and L. Rapanotti. A reference model for requirements engineering. In *Requirements Engineering Conference, 2003. Proceedings. 11th IEEE International*, pages 181–187, 2003. TY - CONF.

[14] R. J. Hall. Feature interaction in electronic mail. In M. Calder and E. H. Magill, editors, *Feature Interactions in Telecommunications and Software Systems VI*. IOS Press, Glasgow, Scotland, UK, 2000.

[15] J. Hammond, R. Rawlings, and A. Hall. Will it work? In *RE '01: Proceedings of the 5th IEEE International Symposium on Requirements Engineering*, pages 102–109, Washington, DC, USA, 2001. IEEE Computer Society.

[16] J. D. Hay and J. M. Atlee. Composing features and resolving interactions. In *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 110–119, New York, NY, USA, 2000. ACM Press.

[17] I. Hsi and C. Potts. Studying the evolution and enhancement of software features. In *16th IEEE International Conference on Software Maintenance (ICSM'00)*, pages 143–151, 2000.

[18] A. Ireland, B. J. Ellis, A. Cook, R. Chapman, and J. Barnes. An integrated approach to high integrity software verification. *Journal of Automated Reasoning: Special Issue on Empirically Successful Automated Reasoning*, 36(4):379–410, 2006. A longer version is available as School of MACS, Heriot-Watt University Technical Report HW-MACS-TR-0027.

[19] M. Jackson. *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices*. ACM Press, 1995.

[20] M. Jackson. *Problem Frames*. ACM Press & Addison Wesley, 2001.

[21] M. M. Lehman and J. F. Ramil. Software evolution: Background, theory, practice. *Information Processing Letters, 2003*, 2003.

[22] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346, 2002.

[23] B. Nuseibeh. Weaving together requirements and architectures. *Computer*, 34(3):115–117, 2001.

[24] V. Rajlich. Modeling software evolution by evolving interoperation graphs. *Ann. Softw. Eng.*, 9(1-4):235–248, 2000.

[25] V. Rajlich. Changing the paradigm of software engineering. *Commun. ACM*, 49(8):67–70, 2006.

[26] L. Rapanotti, J. G. Hall, M. Jackson, and B. Nuseibeh. Architecture-driven problem decomposition. In *12th IEEE International Conference on Requirements Engineering (RE 2004)*, pages 80–89, 2004.

[27] M.-O. Reiser and M. Weber. Managing highly complex product families with multi-level feature trees. In *IEEE International Conference on Requirements Engineering (RE'06)*, pages 146–155, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[28] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature diagrams: A survey and a formal semantics. In *IEEE International Conference on Requirements Engineering (RE'06)*, pages 136–145, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[29] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley & ACM Press, 2000.

[30] C. R. Turner, A. Fuggetta, L. Lavazza, and A. L. Wolf. A conceptual basis for feature engineering. *Journal of Systems and Software*, 49(1):3–15, 1999.