A framework for developing feature-rich software systems

Conference or Workshop Item

A framework for developing feature-rich software systems*

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 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 [33]. Incremental development of new features in feature-rich software, which we may be called *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. Introduction of new features often 'breaks' functioning software, typically because the composed system violates requirements of individual features and/or the composed system exhibits some unexpected new behaviour [9, 18, 2]. Therefore, feature-based development introduces new and important challenges that potentially threaten 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 [37, 12], product-line engineering [7, 4] and software evolution approaches to software development [28, 32, 13, 6].

However, the notion of feature-based development recognized in this paper may be distinguished 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 some 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

*This paper is dedicated to the memory of Dr. Peter Amey at Praxis High Integrity Systems (UK), who passed away on 3rd April 2008. Peter supported the project within which this work was undertaken, serving on the project's Industrial Advisory Board and guiding the authors of this paper towards the SPARK Examiner case study.

integrity software in Section 5. Section 6 highlights the lessons learnt and provides concluding remarks.

2 Related Work

Turner et al. [38] 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 [19]. Jackson [24] formalizes the relationships between major artifacts in software development, which is further developed by Gunter et al [14] and later Hall and Rapanotti [16] to provide specific obligations for stakeholder groups. The conceptual framework for feature-based development proposed in this paper is based on the original work by Jackson [24].

There are a number of RE approaches that can be used to investigate problem structures. The goal-oriented approach KAOS refines high-level goals into sub-goals and then into operational requirements [39]. Operational requirements are then assigned to agents in the solution space [29]. Using a similar notion of goals and goal-refinement, the NFR framework discusses how goals may contribute to achieving software quality [11]. The i* approach to RE, based on an earlier work by Yu and Mylopoulos [40], attempts to augment specifications with contextual information such as the business model [1]. The SCR (Software Cost Reduction) approach describes requirements in a tabular format which captures relationships between controlled and monitored variables [21].

In this paper, we chose the the language and techniques of the Problem Frames approach (PF) [25] to describe problem structures because (i) it provides mechanisms to consider solution components in the problem space [17], (ii) it allows known solution structures to influence problem decomposition [34] and (iii) it can recompose subproblems using a composition operator [26, 27]. PF has also been used to capture change in socio-technical systems [8], but in this paper we are interested in changing software systems.

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" [22]. Schobbens et al. [36] survey feature diagrams and propose a formal semantics of their diagrams. Chen et al. [10] show how feature analysis can help address cross-cutting concerns in architecture, whilst Reiser and Weber [35] 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 [9] and other application domains such as email and web services [18]. Hay and Atlee [20] treat feature interactions as a more general software problem and discuss a feature composition approach to resolve them.

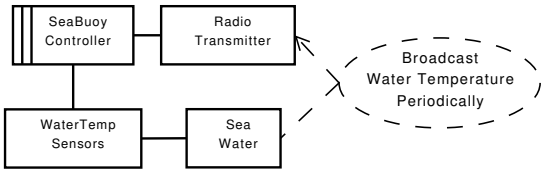


Figure 1. A Simple Problem Structure of a Sea Buoy Controller

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 in order to broadcast temperature information? If one does not know the problem structure in this case, it is difficult to write a specification for the controller.

This example highlights the need for understanding relationships between software development artifacts. Jackson [24] 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. In practice,

descriptions of the requirements and the world context are given by the customers. A *problem*, in this view of requirements engineering, is the challenge of obtaining a correct specification.

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: it means that the artifacts (W, R, S, P and C) may be described in varying degrees of formality, from startchart and temporal logic to natural language. Likewise, showing that an entailment relationship holds for some given artifacts also may be done to different degrees of formality, from mathematical proofs to informal arguments, depending on the description language chosen and the specific needs of the stakeholders. When formal description languages are used, the proof can be done through logical deduction.

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 issue. The next section discusses how this issue is tackled in the context of the framework above.

4 Feature-based Development

In a typical feature-based development project, there is an existing solution that satisfies 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 property in feature-based development. Again the argument for this property may be either formal or informal.

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 [31, 34], we have established that there are

advantages in letting solution structures influence problem structures. For instance, if the developers know that a complex problem can be solved using the Model-View-Controller (MVC) pattern, the problem maybe decomposed in such a way that the subproblems map to components of MVC. 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 $\mathrm{S}_{build-x}$ is a specification for an intermediate build and $\mathrm{W}_{build-x}$ is the state of the world (for $\mathrm{P}_{build-x}$) between P_{now} and P_{future}, a simple reformulation of the entailment relationships (4) gives an important obligation for developers during these steps:

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

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) may not hold between releases because not all properties of W_{future} have been considered, and (6) ensures that each accepted build does not break the software, i.e. R_{now} is still satisfied. R_{now} is therefore satisfied during builds between releases (by the entailment relationship 6), and on each release (by the entailment relationship (4)). Setting this basic 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 some cases, this can be done using formal proofs, and in other cases using evidence, such as test results. The entailment relationships (3) and (5) are other obligations when the 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 next section reports on an experience of introducing a new feature to a critical software system. We focus on how the framework introduced above can be instantiated in practice and how various obligations required by the framework can be discharged.

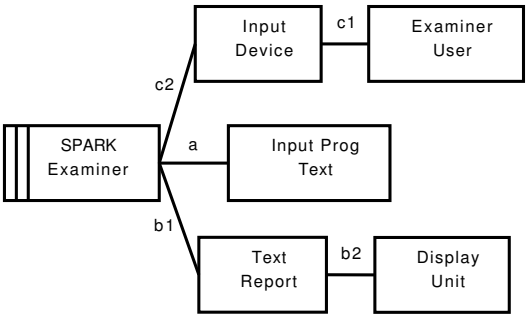
5 SPARK Examiner: An Industrial Example

Designed for high integrity software systems, SPARK is a programming language that uses a subset of standard Ada. The aim of SPARK and the tools supporting it is to enable software developers implement mission-critical applications that are free from run-time errors, such as buffer overflows.

A SPARK program, written in the subset of Ada, is typically annotated with its specification, written as Ada comments. Ada compilers ignore these comments but SPARK tools use them to do various analyses. The annotations embedded in the programs defines the interface (or the "contract") between the components. The definition typically include the signature, the input and output parameters, names of global variables accessed by the component, the pre- and post-conditions, dependencies between variables and so on.

SPARK Examiner, or simply the Examiner, is a tool that performs two checks statically (i.e. before the program executes): (a) the syntax of the SPARK program and (b) the consistency between the code and embedded annotations by means of control, data and information flow analysis. By analyzing the annotations, the Examiner automatically generates verification conditions (or potential theorems) that need to be proved in order to show that the program correctly implements its specification. These theorems can be proved either entirely by the human or with the help of other SPARK tools. For an assignment statement, for instance, the Examiner may generate a verification condition that assigning to the variable a value outside the range defined in the variable declaration is impossible. If this theorem cannot be proved, the program is thought not to be reliable [5, 15, 23]. The Examiner itself is written in SPARK, and therefore subject to its own analysis. The tool has about 120K lines of code.

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 the Examiner. We are, therefore, concerned with the development of the Examiner, rather than its application in the development of other software systems. The Examiner feature used in the discussion is real, but simple enough to illustrate how



a: SE!{getProg} c2: ID!{commands}
c1: EU!{keystrokes} b2: TR!{displayText}
b1: SE!{makeTextReport}

Figure 2. Annotated Context Diagram for Current Examiner

it was developed without inviting unnecessary implementation details.

5.1 Current Examiner

The current release of the 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 [25]. In terms of the artifacts of Sections 3 and 4, they are W_{now} and S_{now} respectively. (Notice we do not show the requirements in context diagrams, but they are shown in problem diagrams.)

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, the user of the Examiner software, the keyboard, the computer file containing a SPARK program that the user wishes to have the program analyzed, the text of the analysis reports for the program (such as error messages and warnings), and the computer display device on which the text report is displayed. The descriptions of the problem world domains are implicit: i.e. through their correspondence with program variables. For instance, the behavior of Examiner User maybe described through the constraints on variables the user controls.

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

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. (Notice that labels such as ID and EU are abbreviation of the relevant problem world domain names.)

The machine domain of the Examiner, or S_{now}, is represented by the rectangle with two vertical stripes Fig. 2. As discussed, the specification S_{now} is described as SPARK annotations within the tool.

There are, of course, several requirements the Examiner satisfies. Without enumerating these requirements, R_{now} of the Examiner can be expressed informally as follows:

> *When the user types in commands through the keyboard, the Examiner checks the syntax of the command, and if the command is meaningful, the 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 error messages.*

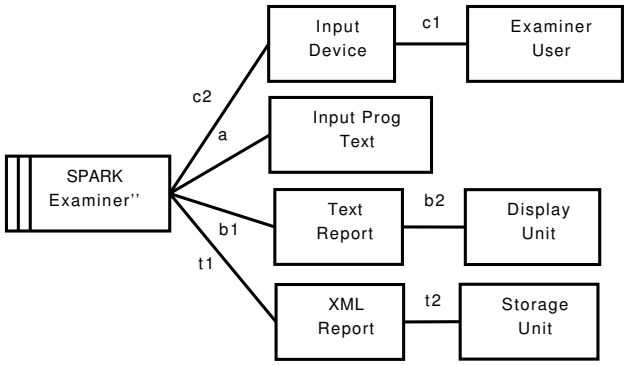
When the descriptions of the world domains, and specification of the machine in Fig. 2 are taken together, they satisfy the above requirements (R_{now}). This is done in two steps. When discharging the developer side obligation (2) of the Examiner, the tool generates its own verification conditions, allowing to prove itself. Discharging the requirements side obligations (1) cannot be done entirely formally: a combination of rigorous testing and customer feedback are used.

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, the 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 invoked by issuing a* `/xml` *switch in the command to the SPARK Examiner.*

The context diagram for the new Examiner that would include the XML report feature can be described as shown



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

Figure 3. Annotated Context Diagram for Future Examiner

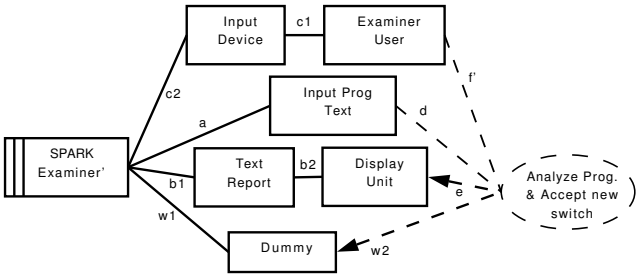
in Fig. 3. The new domains XML Report and Storage Unit in the diagram represent the program analysis result prepared according to a certain XML format, and the disk unit onto which the XML report is to be stored. Notationally, the diagram is unorthodox in one particular sense: with the identity of the 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 to whom this feature is not provided.

5.3 Introducing the New Feature

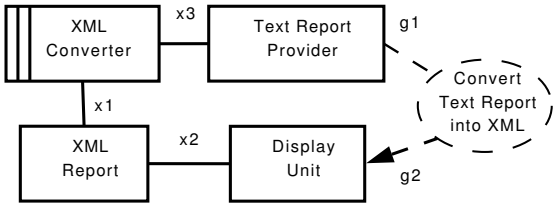
The current Examiner in Fig. 2 does not recognize the command switch /xml to generate the XML report. In the first build towards implementing the new feature, developers attempted to make the Examiner accept the new switch. The first subproblem can therefore be described informally as:

> *When the user issues the new switch, the system acknowledges acceptance by causing a change in the dummy domain (such as a textfile), whilst preserving the existing behavior.*



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

Figure 4. Problem diagram for subproblem 1: Accept the new switch

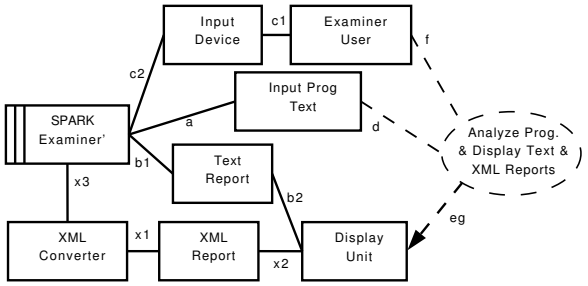


g2: DU!{XMLRep} g1: TP!{TextRep}
x2: XR!{displayXML} x1: XC!{makeXMLReport}
x3: TP!{DummyTextReport}

Figure 5. Problem diagram for subproblem 2: Convert Text to XML

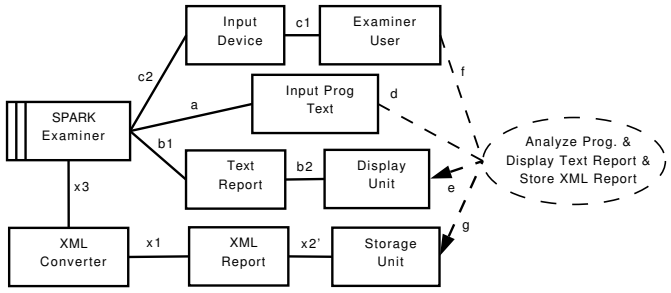
The reason for choosing to tackle this subproblem first is to define the contract between the Examiner and the component implementing the new feature. When implementing this new component, developers will have the knowledge that the new component would recompose with the would-be SE because tool will accept the new switch. However, this change in the structure of SE does not alter the existing behavior in unspecified 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 the Examiner User and Input Prog Text – represented by dotted lines without arrowheads, and observes



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

Figure 6. Problem diagram for subproblem 3: Display text and XML reports



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

Figure 7. Problem diagram for subproblem 4: Final Increment

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 can proof the correctness of the code to show that the modified software is still free of errors. This is in accordance with the entailment relationship (6). If developers wanted to gain further confidence, they would run test cases and even release this version of software (which could include other unrelated and visible changes) and evaluate customer feedback. When developers are confident that the initial redesign has not resulted in the system behaving in unexpected ways, they proceed 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:

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 final 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 entailment relationships (3) and (5).

5.4 Summary

Development practices that we observed in the implementation of the XML and several other features in the Examiner are in line with the systematic framework suggested in Section 4. First, P_{now} was modified, rather than treated

as a black-box. In terms of honoring the obligation during the increment, developers perform a combination of both formal proofs, testing and evaluation of customer feedback. For the correctness of the tool after each build, (4) and (6) can be proved formally. Similarly, the correctness of the tool with respect to its new specification (5) can also be proved formally. To show that the the tool with the new feature satisfies the requirements, developers use test cases and often customer feedback, confirming the suggestion by our framework that a program that satisfies its specification may not necessarily be the program required by the customer.

6 Lessons Learnt and Conclusions

In this paper, we have described a particular type of software development in which existing software needs to be modified and extended to satisfy evolving needs of customers, called 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 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, and for a systematic way of discharging obligations on various stakeholders provides a good foundation for developing dependable software systems, and

- specific obligations for developers and requirements engineers during and at the end of each software release are helpful in 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 [3, 30] 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

This paper has benefited from the helpful comments of our colleagues at The Open University and anonymous reviewers. We thank the late Peter Amey of Praxis High Integrity Systems for 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] F. M. R. Alencar, J. Castro, G. A. C. Filho, and J. Mylopoulos. From early requirements modeled by the i* technique to later requirements modeled in precise uml. In *WER*, pages 92–108, 2000.

[2] D. Amyot and L. Logrippo, editors. *Special issue: Directions in feature interaction research*, volume 45(5) of *Computer Networks*. Elsevier North-Holland, Inc., New York, NY, USA, 2004.

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

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

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

[6] 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.

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

[8] J. Brier, L. Rapanotti, and J.Hall. Towards capturing change in socio-technical systems requirements. In *Proceedings of the 11th International Workshop on Requirements Engineering - Foundation for Software Quality*, pages 225–237, 2005.

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

[10] 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.

[11] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Publishing, 2000.

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

[13] 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.

[14] 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.

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

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

[17] J. G. Hall, M. Jackson, R. C. Laney, B. Nuseibeh, and L. Rapanotti. Relating software requirements and architectures using problem frames. In *Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*, pages 137–144. IEEE Computer Society, 2002.

[18] 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.

[19] 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.

[20] 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.

[21] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Trans. Softw. Eng. Methodol.*, 5(3):231–261, 1996.

[22] 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.

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

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

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

[26] R. Laney, L. Barroca, M. Jackson, and B. Nuseibeh. Composing requirements using problem frames. In *Proceedings of 12th IEEE International Conference Requirements Engineering (RE'04)*, pages 122–131. IEEE Computer Society, 2004.

[27] R. C. Laney, T. T. Tun, M. Jackson, and B. Nuseibeh. Composing features by managing inconsistent requirements. In L. du Bousquet and J.-L. Richier, editors, *ICFI*, pages 129–144. IOS Press, 2007.

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

[29] E. Letier and A. van Lamsweerde. Agent-based tactics for goal-oriented requirements elaboration. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, pages 83–93, New York, NY, USA, 2002. ACM Press.

[30] 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.

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

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

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

[34] 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.

[35] 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.

[36] 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.

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

[38] 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.

[39] A. van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proceedings of Fifth IEEE International Symposium on Requirements Engineering, 2001*, pages 249–262, 2001.

[40] E. S. K. Yu and J. Mylopoulos. Understanding "why" in software process modelling, analysis, and design. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 159–168, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.