Proceedings

Feature Interaction in Composed System

ECOOP 2001 Workshop #08
http://i44w3.info.uni-karlsruhe.de/~pulvermu/workshops/ecoop2001/

In Association with the
15th European Conference on Object−Oriented Programming

Budapest, Hungary −− June 18 − 22, 2001
http://ecoop2001.inf.elte.hu/

Elke Pulvermüller, Universität Karlsruhe
Andreas Speck, Intershop Research
James O. Coplien, University of Manchester
Maja D'Hondt, Vrije Universiteit Brussel
Wolfgang DeMeuter, Vrije Universiteit Brussel
(Eds.)

Universität Karlsruhe
Fakultät für Informatik / Institut für Programmstrukturen und Datenorganisation (IPD)
Adenauerring 20a
76128 Karlsruhe, Germany

Universität Karlsruhe
Fakultät für Informatik
Interner Bericht (Internal Report)

Technical Report No. 2001−14
September 2001

Preface

The history of computer science has shown that decomposing software applications helps managing their complexity and facilitates reuse, but also bears challenging problems still unsolved, such as the assembly of the decomposed features when non−trivial feature interactions are involved. Examples of features include concerns or aspects, black−box or white−box components, and functional and non−functional requirements. Approaches such as object−oriented and component−based software development, as well as relatively new directions such as aspect−oriented programming, multi−dimensional separation of concerns and generative programming, all provide technical support for the definition and syntactical assembly of features, but fall short on the semantic level, for example in spotting meaningless or even faulty combinations. At previous ECOOPs, OOPSLAs and GCSEs dedicated events have been organised around the aforementioned technologies, where we experienced a growing awareness of this feature interaction problem. However, feature interaction is often merely dismissed as a secondary problem, percolating as an afterthought while other issues are being addressed.

This workshop intended to be the first co−ordinated effort to address the general problem of feature interaction in composed systems separately from other issues. Feature interaction is an issue of research in telecommunication since several years. This FICS workshop extends the research to the area of components and composition.

13 contributions give an overview about current research directions in the field of feature interaction. These embrace, for instance, feature description, feature detection, feature composition, features in product lines and logic models supporting the detection and resolution of feature interferences.

The web page of the workshop as well as the contributions to this proceedings may be found at URL: http://i44w3.info.uni−karlsruhe.de/~pulvermu/workshops/ecoop2001/index.html

We would like to thank the program committee for their support as well as the authors and participants for their high−quality submissions and engaged contributions during the workshop.

The FICS Organisers
Elke Pulvermüller, Andreas Speck, James O. Coplien, Maja D'Hondt, Wolfgang DeMeuter

Program Committee

Organisation

Elke Pulvermüller
Universität Karlsruhe, Germany
Email: pulvermueller@acm.org
WWW: http://i44w3.info.uni−karlsruhe.de/~pulvermu/

Andreas Speck
Intershop Research, Germany
Email: a.speck@intershop.com

James O. Coplien
University of Manchester Institute of Science and Technology

Maja D'Hondt
Vrije Universiteit Brussel
Email: mjdhondt@vub.ac.be
WWW: http://prog.vub.ac.be/

Wolfgang De Meuter
Vrije Universiteit Brussel
Email: wdmeuter@vub.ac.be
WWW: http://prog.vub.ac.be/

Table of Contents

Product Lines

Logic Models

Position Paper:
Feature Interaction in Composed Systems

E. Pulvermüller[1] A. Speck[2] J. O. Coplien[3] M. D'Hondt[4] W. DeMeuter[4]

[1]Institute for Program Structures and Data Organization
Universität Karlsruhe, Germany.
http://i44w3.info.uni-karlsruhe.de/~ pulvermu
pulvermueller@acm.org

[2]Intershop
Jena, Germany
http://www-pu.informatik.uni-tuebingen.de/users/speck
a.speck@intershop.com

[3]Bell Laboratories Lucent,
Naperville IL, USA
http://www.bell-labs.com/~ cope/
cope@research.bell-labs.com

[4]Vrije Universiteit Brussel, Belgium
http://prog.vub.ac.be/
mjdhondt@vub.ac.be
wdmeuter@vub.ac.be

Keywords: Feature interaction, feature, feature modeling, composition

Abstract

Feature interaction is nothing new and not limited to computer science. The problem of undesirable feature interaction (feature interaction problem) has already been investigated in the telecommunication domain. Our goal is the investigation of feature interaction in component-based systems beyond telecommunication. The position paper outlines terminology definitions. It proposes a classification to compare different types of feature interaction.

A list of examples give an impression about the nature and the importance of feature interaction.

1 Introduction and Problem

The workshop "Feature Interaction in Composed Systems" deals with a problem which is not new. In fact, as opposed to that, it's a problem even older than human life.

In the domain of telecommunication this problem was explicitely explored first in the beginning 1990s using the term "feature interaction problem". In a series of workshops (the first was held in 1992) the difficulty to manage implicit and unforeseen interactions between newly inserted features and the base system have been examined.

However, the problem is not limited to the telecommunication domain. As opposed to that

feature interaction is an issue which occurs in nearly all domains although not known under the name “feature interaction”.

In the last two years we found a growing awareness of this problem in the domain of system composition far beyond telecommunication issues. With the emerge of aspect-oriented programming it has become obvious that with the growing number of system units their interaction is a problem of its own, requiring research by its own. While AOP, CF, SOP, AP [1] and related approaches reveal this problem it’s not limited to those approaches either. It already exists in object-oriented or component-oriented systems, for instance.

In many discussions we found that already a lot of work exists dealing with feature interaction in various domains and applying various programming paradigms.

The workshop aims at collecting the different problems and to provide a platform for knowledge transfer.

Some important questions are:

- What are suitable definitions for “feature” and “feature interaction”?
- Are there any commonalities in the different problems and / or their solutions?
- How can the problems be categorised in problem classes?
- What is the influence of advanced separation of concerns or component-based approaches to feature interaction problems and vice-versa?

While it is easy to compose a system technically it’s hardly explored how systems can be combined in a way that the result is a valid and also reasonable system. The developer faces a lack of clearly structured composition and connection concepts.

2 Terminology or “What is a Feature?”

When discussing about feature interaction it’s important to consider existing terminology.

In a series of workshops the feature interaction problem has already been investigated in the telecommunication domain since 1992. The workshop statement explains that “feature interaction occurs when one telecommunication feature modifies or subverts the operation of another one”.

A definition found in the telecommunication community is as follows: “The feature interaction problem can be simply defined as the *unwanted interference between features running together in a software system.*” A simple example given in [9] is a mailing list echoing all emails to all subscribers. If one of the subscribers has enabled the vacation program (without first suspending messages from the mailing list) an infinite cycle of mail messages between the mailing list and the vacation program will occur.

In [10] a feature is defined as “an extension to the basic functionality provided by a service” while a service is explained as “a core set of functionality, such as the ability to establish a connection between two parties”.

While these definitions concentrate on telecommunication, large and distributed systems or multimedia systems in particular the goal of this workshop is the investigation of feature interaction in the domain of software composition and systems built from components.

We distinguish the terms “feature”, “concern” and “requirement”, “service”, “component” and “aspect” according to our experiences as follows:

In [4] you may find the following explanation for “features”: “A feature is something especially noticeable: a prominent part or detail (a characteristic). A feature is a main or outstanding attraction, e.g. a special column or section in a newspaper or magazine”. Its origin is from Latin: “factura” which means the “act of making” or from

"facere" which means "to make, do".

According to this definition we use the term feature in a broader sense than just as an extension to some basic functionality. It's an observable and relatively closed behaviour or characteristic of a (software) part. In software, it's not just an arbitrary section in the code except this code section "makes something of outstanding attraction". This definition is fuzzy revealing the fuzzy nature of features. This viewpoint is consistent with that expressed in [2] where a feature is explained as "any part or aspect of a specification which the user perceives as having a self-contained functional role".

A feature is something an application has to do. It may be composed from other features. A basic feature should be as small as possible (basic building blocks). What a feature does should be documented. This might be done either manually (the implementer is forced to document it) or by deriving it from the program structure and program flow.

We have a distinction between problem domain features and features on the implementation level (cf. figure 1). Different alternative implementation features may realise the higher-level problem domain features. Moreover, a problem domain feature may be implemented by one or more implementation feature.

A component may have several features and, vice-versa, it realises at least one feature (otherwise the component is not useful). Moreover, a component implements functionality and has non-functional properties (e.g. real-time properties, platform). "Non-functional" is all which is not implemented directly. Therefore, a component *has* non-functional properties but not *implements* those. Non-functional properties are additional properties implicitly resulting from the code. However, a consequence is that the developer may have to implement mechanisms to check whether the required non-functional requirements are met, i.e. you may even find the non-functional properties in the code.

A feature may be implemented as functionality

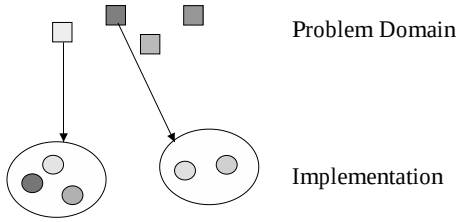


Figure 1: Levels of Features

or may have a non-functional nature. The term "feature" captures both.

A feature has the property that it is a service if it is localised in one component and if it refers to some functionality. However, a feature may be implemented in several components. In this case, the feature is (implemented) cross-cutting. This is more a question of how a feature may be implemented than a question of the nature of the feature itself. Aspects are a notation which may be used to express cross-cutting features (primarily on the implementation level). Therefore, the concept of aspects is orthogonal to the nature of features.

A requirement is something a stakeholder demands and it refers to the problem domain whereas a feature is not limited to the problem domain. A requirement may result in one or several feature(s) in the final system.

A concern is something we are concerned about (at the moment). Note that this is a temporal statement while a feature is permanent. A feature might become a concern if somebody is concerned about. On the other hand, a developer or stakeholder may be concerned about something which is not a feature. Therefore, not every concern results in a feature.

We have to distinguish between intended interactions between features, interactions between features which is not intentional but don't result in errors (or may even have positive side-effects) and unintended and undesirable feature interaction not known in advance and leading to faulty applications. Figure 2 shows a classification scheme al-

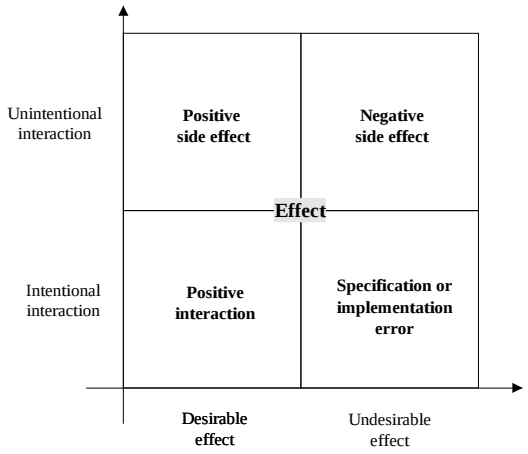


Figure 2: Feature Interaction Classification

lowing to classify, compare and assess a detected interaction.

3 Examples for Feature Interaction (Problems)

In the following some examples about undesirable or unforeseen feature interactions are listed. These examples are of different application domains giving an impression about the range of occurrences in practice.

- Example with modularised Corba functionality [5]

  In order to keep an application independent from the communication technique the communication code may be separated in aspects applying aspect-oriented programming.
  When a client wants to access a service exposed by a specific server the client has to obtain an initial reference to the server. This can be done either via a name server or via a file-based solution (the reference is stored as a string in a file which is accessible for clients and server). Aspects realising one of these two alternatives are exclusive. This is already known at design and implementation time. Let us assume that this knowledge was not captured at design time. As a consequence it might happen that the developer configures the system during the deployment phase with both mutual exclusive features. It might happen that even the compilation or weaving doesn't report this as an error. However, the running system behaves in an unforeseen way.
  An approach to deal with this problem may be found in [3]. Logical rules describe the dependencies between the aspects. During run-time pre- and post-conditions assure that these logical rules are not violated.

- Telecommunication

  Feature interaction is a typical problem in the telecommunication domain. Due to high competition and market demand telecommunication companies are urged to realise a rapid development and deployment of features.

  A list of features in the telecommunication domain may be found at [8]. Examples of features are forwarding calls, placing callers on hold, or blocking calls. It's obvious that some of the features lead to effects which are unforeseen if combined.

  There are multiple approaches to deal with the problem in the telecommunication domain. In the mentioned workshop series [7] a platform is provided for exchanging solutions in practice and theorie.

- Medicine and human body

  Feature interaction is well-known in the context of health. For medicaments a common approach is to provide a standard documentation (instruction leaflet) about the ingredients, the application and it also lists the known (potential) side-effects and interactions with other medicaments or parts of the

human body. These side-effects are more or less dangerous. The lists are developed by means of experiments during the development of the medicaments and by means of experiences and observations afterwards.

An example of a positive side-effect is a medicament called aspirin developed to be used against headache. Experiences and research proved that this medicament affects the blood-picture in a way that it lowers the danger of a cardiac infarction.

- Elevator configuration as described in [6]

  In [6] a system called VT is described which configures elevator systems at the Westinghouse Elevator Company. An elevator has cables which have some weight. This weight influences the traction ratio needed to move the car. The traction ratio influences the cable equipment (also the cable weight). Therefore, we have a circular feature dependency. In case we would like to improve the security standards and therefore increase the cable quality (which results in a higher cable weight) we have an interaction with the traction ratio which might be unforeseen if this dependency is not specified and documented. VT uses artificial intelligence (propose-and-refine approach) to deal with this problem. Individual design parameters and their inferences are represented as nodes in a network.

4 Summary and Conclusion

Feature interaction is an issue in different domains (not limited to computer science even). Due to the complexity it's usually impossible to foresee all potential interactions of the different features within one system. However, it is possible to approach the problem by identifying as many as possible unforeseen (and maybe undesirable) interactions.

Software features may be modeled by means of a suitable notation similar as designs may be modeled using UML. Notation may be found in the domain engineering discipline [1]. Even the interactions or dependencies, respectively, of features may be modeled. Incompatible combinations or default combinations may be defined already during the domain analysis phase.

In this paper we approached the terminology and proposed a classification scheme to compare different feature interaction types.

Starting with this workshop we aim at building a catalogue of problems and potential solutions. Although it is not expected that there will be one best, exact and general solution (approximative) solutions may exist for certain problem domains or specific systems. Research results in the telecommunication domain are expected to be helpful also for systems built from components.

From this catalogue we aim at an improved classification allowing to compare different interaction types or problems and a comparative catalogue of solutions. We aim at unifying problems and solution approaches.

References

[1] K. Czarnecki and U.W. Eisenecker. *Generative Programming - Methods, Tools, and Applications*. Addison-Wesley, 2000.

[2] ESPRIT Working Group 23531, `http://www.dcs.ed.ac.uk/home/stg/fireworks/workshop.html`. *FIREworks, Workshop on Language Constructs for Describing Features.*, 2001.

[3] H. Klaeren, E. Pulvermüller, A. Rashid, and A. Speck. Aspect Composition applying the Design by Contract Principle. In *Proceedings of the GCSE'00, Second International Symposium on Generative and Component-Based Software Engineering*, LNCS, Erfurt, Germany, September 2000. Springer.

[4] Merriam-Webster OnLine, http://www.m-w.com/. *Merriam Webster's Collegiate Dictionary*, 2001.

[5] E. Pulvermüller, H. Klaeren, and A. Speck. Aspects in Distributed Environments. In K. Czarnecki and U. W. Eisenecker, editors, *Proceedings of the GCSE'99, First International Symposium on Generative and Component-Based Software Engineering*, LNCS 1799, Erfurt, Germany, September 2000. Springer.

[6] M. Stefik. *Introduction to Knowledge Systems*. Morgan Kaufmann Publishers Inc., 1995.

[7] University of Glasgow, http://www.cs.stir.ac.uk/ mko/fiw00/. *Feature Interaction Workshop*, 2001.

[8] University of Glasgow, http://www.dcs.gla.ac.uk/research/hfig/features.html. *The Feature List*, 2001.

[9] University of Strathclyde, http://www.comms.eee.strath.ac.uk/~fi/. *Feature Interaction Group*, 2000.

[10] University of Waterloo, http://se.uwaterloo.ca/~s4siddiq/fi/fip.html. *Feature Interaction Problem*, 2001.

Modelling Feature Interactions in Mobile Phones

Louise Lorentsen
University of Aarhus and Nokia Research Center (visiting),
ext-louise.lorentsen@nokia.com
Antti-Pekka Tuovinen, Jianli Xu
Nokia Research Center, P.O.Box 407, FIN-00045 NOKIA GROUP, Finland
antti-pekka.tuovinen@nokia.com, jianli.xu@nokia.com

Abstract

A modern mobile phone supports many features: voice and data calls, text messaging, phonebook, calendar, WAP browsing, games, etc. All these features are packaged into a handset with a small display and a special purpose keypad. The limited user interface and the intertwining of logically separate features cause problems in the development of the UI software for mobile phones. In this paper, we look at the problem of feature interaction in the UI of Nokia's handsets. We present a categorization of feature interactions and describe our approach to modelling feature interaction patterns that uses explicit behavioural models of features and interactive graphical simulation. We use Coloured Petri Nets as the modelling language.

1 Introduction

The context of this work is the development of the UI software for Nokia's mobile phones. In this domain, the term *feature* means functionality of the phone that is accessible or visible to the user via the UI and implemented by software. The features are implemented by UI applications in the proprietary mobile phone UI software system. *Feature interaction* means a dependency or interplay of features. The interactions can be conceptually simple usage dependencies or more complex combinations of independent behaviours.

The development of the user interface software for mobile phones is a concurrent and highly distributed engineering process. There is also strong pressure for reusing SW components in as many products as possible. In this kind of environment, it is important to identify and clearly specify the right interactions between the separate features of the mobile phone at an early stage of the development. This helps to avoid costly delays in the integration phase of a set of independently developed features. Precise descriptions of the interactions are also needed when planning the testing of the UI software. The number and type of interactions that a feature has with other features are also indicators of the cost of developing the feature.

Currently, feature interactions are not systematically documented. Often the most complex interactions are not fully understood before the features are first implemented. The goals of this work are to *identify categories of interactions* that are specific to the domain and to create behavioural models that *capture the typical feature interaction patterns* in each category.

The heart of our approach is an executable behavioural model of the underlying UI architecture and the individual features. As the modelling language we use Coloured Petri Nets, a visual, both action and state-based specification formalism that is suitable for modelling concurrent activities and flows in complex systems [3,4,5]. They have precise executable semantics which makes it possible to simulate the behaviour specified by CPN models. The tool that we use makes it possible to add domain specific graphics for visualisation and interaction purposes.

2 Classification of Feature Interactions

Each mobile phone product follows a certain *UI style* that captures the UI design of a product family. It describes the physical structure of the UI and the basic mechanisms of user interaction and it has a relatively long lifetime. The *UI specification* of a product defines the features of the product by showing the UI design and by describing the detailed user interaction for each feature.

Feature interactions come from different sources. Category I of interactions is the *use interaction* between features. For instance, the *task-oriented* nature of the user interface requires that when browsing the phone numbers stored in the phone, a call can be made to a number directly from the browser. This represents an interaction between the 'phonebook' and 'mobile originated call' features that is necessary to deliver a smooth and seamless service to the user. When compared with PC software, the applications in the phone SW have much more these kind of hard-wired dependencies.

Category II comes from the need to *share the limited UI resources* (screen, keypad) between many features that can be activated independent of each other. Because of the prioritization of the users tasks (and the associated features), important events may interrupt less important activities. For example:

- an incoming call *screens* phonebook browsing for the duration of the call but the browser application does not know it.,
- hang-up key *stops* search from phonebook (the browser is killed), and
- an incoming call *suspends* a game but the game is saved and it can be continued.

Category III involves interactions where one feature affects other features by making them unavailable or by *modifying their behaviour* in some other way. For instance, the 'any key answer' feature makes it possible to answer an incoming call by pressing any key on the keypad and the 'key guard' feature locks the keypad for accidental key presses. The combined effect of these features is that if 'any key answer' is enabled and 'key guard' is on, an incoming call can be answered only by pressing the 'send' (off-hook) key. However, once the call is open, 'key guard' is disabled for the duration of the call and then enabled again automatically.

The use interactions are thoroughly specified in the UI specifications and they are not problematic from the implementation point of view. However, the interactions of the categories II and III are much more difficult to manage in software design and implementation; they also lack systematic documentation. Therefore, we concentrate on modelling and documenting the typical feature interaction patterns that belong to the latter two categories.

3 Our Modelling Approach

Coloured Petri Nets (CP-nets or CPN) [3] is a graphical modelling language with a well-defined semantics allowing simulation of the behaviour specified by CPN models as well as formal analysis. In contrast to many other modelling languages, CP-nets are both state and action oriented. CP-nets has proven powerful for modelling of concurrent systems and a number of successful projects have demonstrated its usefulness in modelling and analysis of complex systems, e.g., [1,4,6,8]. We use the tool Design/CPN [7] that supports editing, simulation and validation/verification of CP-nets.

Figure 1 gives an overview of the CPN model by showing how it has been hierarchically structured into 14 modules (also referred to as subnets or pages). Each node in Fig.1 represents a subnet of the CPN model. An arc between two nodes indicates that the source node contains a *substitution transition* whose behaviour is described in the subnet represented by the destination node.

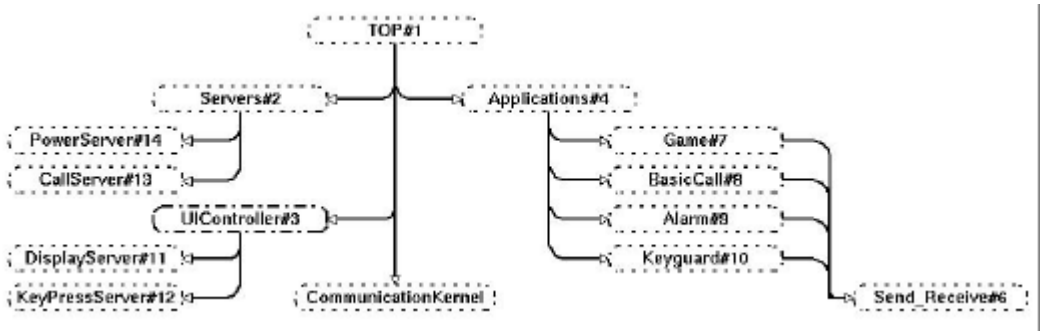


Fig.1. The hierarchy page

The CPN model consists of four main parts corresponding to four concepts of the phone UI software system: applications, servers, UI controller, and communication kernel. *Applications* implement the features. The CPN model presented here includes four features: 'game', 'basic call', 'alarm', and 'key guard' features. *Servers* implement the basic capabilities of the phone. Applications implement the behaviour of features by using the services of servers. The CPN model presented here includes two servers: 'call' and 'power' servers. Applications make the feature available to the user via a *user interface*. Servers which provide the basic capabilities of the applications do not have user interfaces. Servers and applications are communicating by means of asynchronous message passing. The messages are sent through the *communication kernel*.

The subnet Top depicted in Fig.2 is the top-most page of the CPN model and provides the most abstract view of the CPN model. The page consists of four substitution transitions corresponding to the four parts mentioned above. The detailed behaviour of UIController, Servers, CommunicationKernel, and Applications is modelled on subnets associated with the substitution transitions.

A CP-net is created as a graphical drawing with some textual inscriptions. In contrast to many other modelling languages CP-nets are both state and action oriented. A state of a CP-net is represented by means of *places* which are drawn as ellipses with a name positioned inside. The places contain *tokens*, which carry data values, in CPN

terminology referred to as *colours*. Each place has a type, in CPN terminology referred to as a *colour set* which determines the kind of tokens which can reside on the place.

Actions of CP-nets are represented as *transitions* which are drawn as rectangles with a name positioned inside. The transitions and places are connected by *arcs*. Transitions remove tokens from places connected by incoming arcs and add tokens to the places connected by outgoing arcs. The tokens removed and added are determined by *arc expressions* which are textual inscriptions positioned next to the arcs. In the Design/CPN tool, the inscription language is Standard ML.

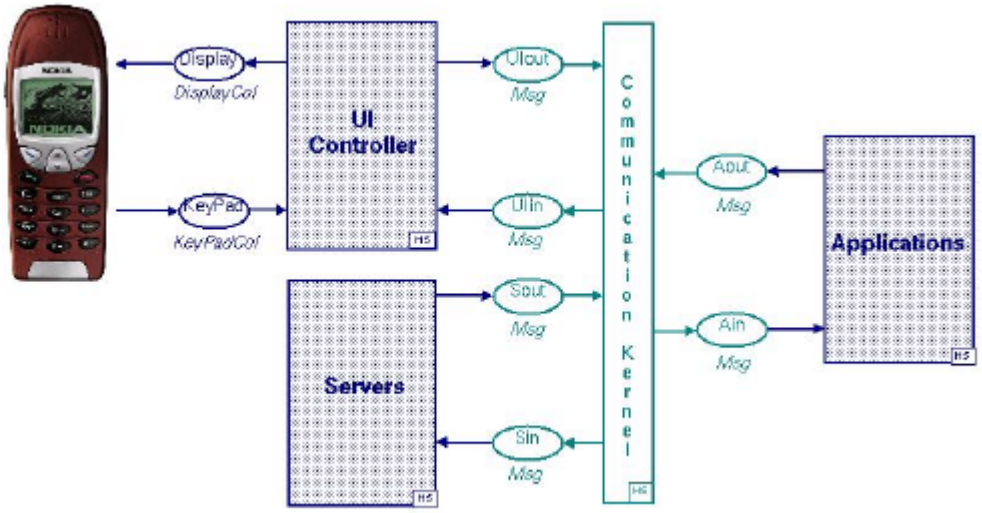


Fig.2. Page Top

The use of substitution transitions allow the user to relate a transition to a more complex CP-net. The idea is analogous to the module concepts found in many programming languages. Furthermore, CP-nets has the concept of *fusion places* which allow the user to specify that a set of places are identical even though they are drawn as individual places (possibly belonging to different subnets). Using these two constructs together with Design/CPN's ability to save and load subnets of a CPN model we have constructed a CPN model of the phone UI software system where features can easily be added and removed. Hence, large parts of the CPN model can be reused when we later model other products with new features.

In addition to a graphical representation, CP-nets have formally defined semantics which makes it possible to simulate the behaviour specified by the CPN model. Design/CPN provides facilities for automatic simulations as well as interactive (step-by-step) simulations. However, the models we create are simulated and discussed in a forum of UI designers and software developers who are not experts in (or even familiar with) CP-nets. An important aspect of our work is therefore to extend the created CPN models with a layer of domain specific graphics which makes it possible to plan and control simulations and get feedback and information from these simulations without interacting directly with the CP-nets.

We have made two extensions to the CPN model that allow the *visualisation* of the current state of the CPN model and the behaviour of the CPN model during simulation. Firstly, the state of the phone as the user observes it on the handset is visualised via an animation of the display. Figure 4a shows a snapshot of the animation taken during a simulation of the CPN model. Secondly, the CPN model is extended with Message Sequence Charts (MSCs) [2] to be automatically constructed as graphical feedback from simulations. We chose to use MSCs in the visualisation because the SW designers already use them.

Fig. 4b shows an example of such a MSC automatically generated from a simulation of the CPN model. The MSC contains a vertical line for each of the applications and servers in the phone UI software system. The arrows between the vertical lines correspond to messages sent in the system. The communication sequence corresponds to a scenario where the mobile phone receives an incoming call while the user is playing a game (an interaction between the 'game' and the 'call' features). The sequence of events in the scenario is:

Fig. 4a Animation of display

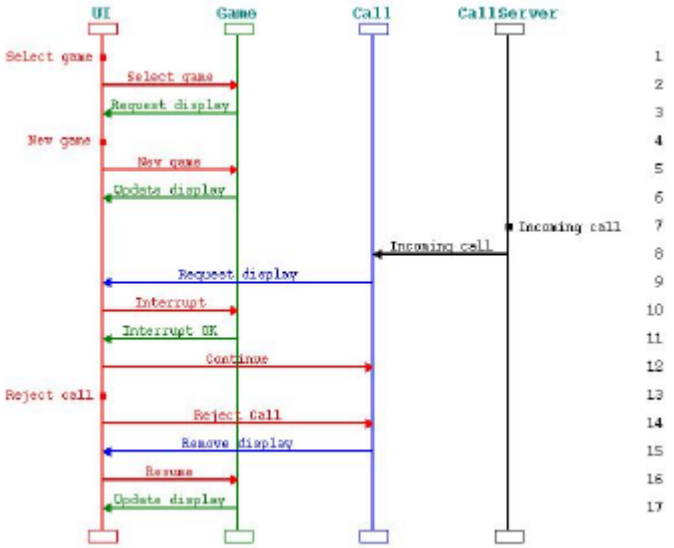
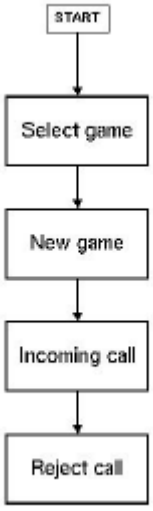


Fig.4b Message Sequence Chart

- The user selects a game from the menu (line 1)
- The game feature is notified and it requests the display (lines 2-3)
- The user selects a new game (line 4)
- The 'game' feature is notified and it changes the contents of the display accordingly (lines 5-6)
- An incoming call arrives. The 'call' server notifies the 'call' feature (lines 7-8)
- The 'call' feature requests the display (line 9)
- The display is currently in use of the 'game' feature. The UI controller interrupts the 'game' feature and after the interruption has been acknowledged the display is granted to the 'call' feature (lines 10-12)

- The user rejects the call (line 13)
- The ‘call’ feature is notified and the display is removed (lines 14-15)
- The ‘game’ feature is resumed (lines 16-17)

Note that in the above scenario the UI controller is responsible for handling the interrupt (lines 10-12) and resume (lines 16-17) of features. The features do not have to know which features they potentially interrupt or are interrupted by. This makes it very easy to add and remove features from the CPN model without changing the subnets modelling the other features.



We have made two extensions to the CPN model to make it possible to *control* the simulations without directly interacting with the CP-nets. The first extension makes it possible to control simulations by clicking the keys of the image of the mobile phone in Fig.4a. The second extension makes it possible to set up a scenario to be simulated. The scenario is specified as an ordered series of events. Figure 5 shows how the scenario corresponding to the MSC in Fig. 4b (where the mobile phone receives an incoming call while the user is playing a game) is specified. In this way it is possible to inspect interesting scenarios without manually pressing the keys of the mobile phone in Fig. 4a.

Fig. 5

4 Summary and Further Work

The current model provides the basic UI infrastructure where we can plug in features. The model provides interactive and automatic graphical simulation. We have already identified some basic interaction patterns in the ‘classic’ UI style of the 6210 phone.

We are now adding more features to the model to build a comprehensive set of interaction patterns. One important task will be to link the interaction patterns to existing implementation patterns. Possible uses of the models include giving ideas to the UI architecture development work and regression testing when changing the logic of the features included in the model.

References

[1] S. Christensen and J.B. Jørgensen. "Analysis of Bang and Olufsens's Beolink Audio/Video System using Coloured Petri Nets". In P. Azéma and G. Balbo, editors, Proceedings of ICATPN'97, volume 1248 of Lecture Notes in Computer Science. Springer-Verlag, 1997.

[2] ITU (CCITT). Recommendation Z.120:MSC. Technical Report, International Telecommunication Union, 1992.

[3] K. Jensen. "Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts". Monographs in Theoretical Computer Science. Springer-Verlag, 1997. ISBN:3-540-60943-1.

[4] K. Jensen. "Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 3, Practical Use". Monographs in Theoretical Computer Science. Springer-Verlag, 1997.

[5] L.M. Kristensen, S. Kristensen, and K. Jensen, "The practitioner's guide to Coloured Petri nets". International Journal of Software Tools for Technology Transfer 2(2): 1998, pages 98-132.

[6] L. Lorentsen and L.M. Kristensen. "Modelling and Analysis of a Danfoss Flowmeter System using Coloured Petri Nets". In M. Nielsen and D. Simpson, editors, Proceedings of ICATPN'00, volume 1825 of Lecture Notes in Computer Science, pages 346-366. Springer-Verlag, 2000.

[7] Design/CPN Online. http://www.daimi.au.dk/designCPN/.

[8] J. Xu and J. Kuusela. "Analyzing the execution architecture of mobile phone software with coloured Petri nets". International Journal of Software Tools for Technology Transfer 2(2): 1998, pages 132-143.

Feature'Interactions Outside a Telecom Domain

Lynne Blair[1], Gordon Blair[1], Jianxiong Pang[2], Christos Efstratiou[2]

[1] *Department of Computer Science, Faculty of Science, University of TromsøN-9037 TromsøNorway.*
Tel: +47 77 64 52 09, Fax: +47 77 64 45 80
[2] *Computing Department, Lancaster University, Bailrigg, Lancaster, LA1 4YR, U.K.*
Tel: +44 (0)1524 593802, Fax: +44 (0)1524 593608
email: {lb, gordon, efstrati}@comp.lancs.ac.uk, j.pang@lancaster.ac.uk

Abstract. Feature interactions in the original sense of the term (i.e. within a telecommunications domain), have now been the subject of significant research activity for over ten years. This paper considers several different sources of interactions in other domains, arising during the course of our research at Lancaster. These interactions are taken from a variety of areas within the field of Distributed Systems, and stand to benefit greatly from the application of techniques developed in the feature interaction community. Furthermore, we believe they represent a potentially important generalisation for feature interaction research.

1. Introduction

The term *feature interaction* can simply be viewed as an "interference between services or features" [Calder00]; more specifically, such an interaction occurs when "the behaviour of one feature is affected by the behaviour of another feature or another instance of the same feature" [Kimbler95]. Several taxonomies have been produced in order to try and classify different types of interaction (including [Cameron94], [Kimbler95] and [Hall98]). A simple, yet we believe helpful, distinction from [Kimbler95] is between:

- interactions that occur because the requirements of multiple features are not compatible, and
- interactions that occur when a feature behaves differently in the presence of other features.

Within the telecommunications domain, there are numerous well-documented cases of feature interactions; for examples, we refer the reader to the series of workshops in Feature Interactions in Telecommunications (and Software) Systems, e.g. [Dini97], [Kimbler98] and [Calder00]. However, recently it has become increasingly obvious that research into methods to detect and resolve such interactions in telecom systems is also of great significance outside the telecom domain. In fact, as recognised in [Calder00], "the subject has relevance to any domain where separate software entities control a shared resource". Furthermore, interactions can often be traced back to the fact that "two 'features' manipulate the same entities in the base system, and in doing so violate some underlying assumptions about these entities that the other 'features' rely on" [Plath98].

In an earlier position paper [Blair00], we describe some interaction problems arising from Internet-based and multimedia/ mobile systems that led to the recently funded "FILBETT" project: Feature Interactions – Life Beyond Traditional Telephony (EPSRC GR/N35939/01). This current paper extends our earlier one by providing new examples of interactions that

[1] Currently on leave from the Computing Department, Lancaster University.

have arisen in research within the Distributed Multimedia Research Group at Lancaster University, in addition to cataloguing a few other 'non-traditional' feature interactions from the literature.

In the remainder of this paper, we first provide brief details of the scope of the FILBETT project below (section 2) and then document some of the 'non-traditional' interactions we have come across (section 3). Finally, we discuss some of the major techniques that exist for the detection and resolution of feature interactions (section 4) and then draw our conclusions (section 5).

2. FILBETT – Life Beyond Traditional Telephony

2.1. Overview

Motivation for this project came from a number of examples of interactions that we encountered when looking at Internet-based and multimedia/ mobile services. As expressed by Heilmeier, "the telecom industry is quickly evolving from 'POTS' (plain old telephone services) to 'PANS' (pretty awesome new services)" [Heilmeier98]. These new services are able to utilise the power of Internet-based (IP-based) and multimedia/ mobile systems and, as the number of new services grows, the potential for interactions between services will inevitably explode.

To help to address this, the main goal of the FILBETT project is to consider various new and emerging types of *feature interaction* that are likely to arise from the increasing popularity of *mobile* systems and services. A secondary, but still important, goal is to consider *IP-based* services and *multimedia* services. We plan to use formal modelling and analysis methods in this work, building on earlier work that we have done. Although the project is still in its early stages, some our initial interaction examples are presented below.

2.2. Some 'non-traditional' interaction scenarios

A number of interactions were identified in our previous position paper; these are listed below, but for details the reader is referred to [Blair00].

- combining a traditional telecommunications service with Internet access
- potential interactions with one-to-many services
- TCP flow-control mechanisms, and protocol interactions in general
- sharing demand for network bandwidth: web browsing and viewing a video stream
- interactions occurring with multipoint conferencing units (MCUs)
- mobile resource interactions concerning bandwidth and power management
- problems with TCP over wireless networks

A number of further interactions have been identified in work at Lancaster on mobile computing (see [Efstratiou00] and [Efstratiou01]). In summary, most of these scenarios concern conflicting *adaptation policies*. In an attempt to maintain an appropriate level of quality of service, many mobile systems employ various adaptation mechanisms. However, problems (interactions) may arise if separate adaptation mechanisms are employed for different attributes. Examples include mechanisms to adapt/ manage power consumption, network bandwidth, proxy behaviour (e.g. in web browsing) and choice of location sensing mechanism. For example, consider a mobile device that employs two independent adaptation mechanisms: one for managing power and the other for managing network bandwidth. If power is running low, the power management mechanism will request applications that are

using network bandwidth to postpone this use, so as to place the network device in sleep mode. However, as a consequence, the network adaptation mechanism will now detect unused bandwidth and will notify applications that they can use this spare bandwidth, in direct conflict with the power management adaptation mechanism!

Note that a further interesting dimension to adaptation is user-configuration of devices whereby a user can express preferences over different adaptation policies depending on his/ her *context*. For example, power management mechanisms may be crucial if the user is working in the field, but less important in the office where an alternative power supply exists.

3. Additional Interaction Scenarios

Whilst FILBETT is primarily concerned with the generalization of feature interactions to a new world of mobile, multimedia and IP-based services, it is clear that the value of research into such interactions does not stop here. Two further areas from our work that would benefit from the application of feature interaction research are described below. As an aside, some additional examples of 'non-traditional' interactions can be found in [Hall00] and [Fireworks97], relating to email systems and a variety of miscellaneous examples (including a lift system, a tape-deck system, a metro ticketing system, etc.) respectively.

3.1. Component-based middleware

At Lancaster, we are interested in component-based middleware platforms such as the CORBA Component Model of CORBA v3, .NET or Enterprise Java Beans. Associated component-based development methodologies focus on the provision of means for specifying individual components together with their composition (i.e. an architecture) [Szyperski98]. By allowing new components to be added, and existing software to be packaged as components, we obtain an incremental development model for evolutionary and dynamic architectures. However, this raises two key questions:

- When we *compose* an architecture, how can we be confident that components work well together, that there are no unwanted or subtle interactions and that the result is coherent?
- When we *adapt* an architecture, how can we be confident that replacements or updates behave as expected, especially in tandem with other components?

Existing component-based methodologies provide little in the way of support for these problems. Typically, architectures are verified in terms of type compatibility between (required and provided) interfaces. In addition, checks may be carried out on the validity of architectures against certain style rules [Shaw96][Medvidovic00]. However, this is not sufficient to capture the more subtle problems associated with unwanted interactions between components. This is an area that would benefit greatly from the application of techniques from feature interaction research. In particular, a *hybrid approach* (see below) would allow *design-time* checks to be carried out on initial architectures and expected variations, whereas *run-time* techniques could be used to discover problems after re-configuration and also to catch problems not foreseen from static analysis.

Adding an extra dimension to this analysis, we are also interested in *reflective middleware* whereby component-based approaches apply not only to the application/ service level, but also to the structure of the middleware itself [Blair98][Blair01]. Reflection is then used to provide introspection and adaptation of this middleware structure via a *meta-level*. This approach enables middleware to be customized for a particular application domain, e.g. a small footprint system for an embedded device, and also to be re-configured if environmental assumptions change, e.g. to change a transport protocol or compression

strategy if now operating over a wireless link. Essentially, this provides the extra capability of being able to *adapt* the non-functional properties of an application (real-time performance, security, availability, etc). Again, it is vital to know if there are any unwanted side-effects of the changes, e.g. does the new availability policy conflict in any way with security requirements).

3.2. Behaviour in Co-operative Virtual Environments

PING is an EU-funded project looking at the development of an object-oriented framework for the support of distributed and co-operative virtual environments, which can then be specialised on a per-application basis (IST-1999-11488). Lancaster University is responsible, along with others, with the modelling of *behaviour* in such virtual environments. The approach is to represent all entities in a virtual environment as passive or active objects. A passive object essentially consists of a set of publicly exposed attributes that can be altered in interaction with other objects; an active object on the other hand also includes behaviour (which in Ping is expressed as a series of reactive scripts written in the Junior scripting language [Boussinot01]). We are investigating an *aspect-oriented* approach to composing such behaviours using Junior, with consideration of many aspects including the capturing of (virtual) world physics (gravity, inertia, etc), distributed systems policies such as replication and consistency management, reaction to collisions, and also autonomous behaviour relating to the object. Furthermore, it is important in PING to be able to adapt behaviour as environmental conditions change, e.g. to minimise event dissemination if operating over a modem. We are investigating an approach whereby monitoring and adaptation will be modelled as further behavioural aspects, resulting in self-adapting object behaviours (c.f. reflection above).

Although this is a rather different application domain, the problems are similar to those considered above. In particular, we are concerned about the initial configuration and the subsequent re-configuration of a platform. In this case however, we are concerned about interaction between behaviours at both an inter- and intra-object level. Importantly, in this work we already have a significant advantage in that Junior has a formal operational semantics (expressed using rewriting rules) [Boussinot00], thus aiding formal analysis.

4. A Brief Summary of Feature Interaction Detection and Resolution Techniques

Existing analysis techniques can be seen to fall into 3 broad categories: *off-line* (or design-time) techniques, *on-line* (or run-time) techniques and *hybrid* techniques [Calder99].

With *off-line* techniques, a model of the base system and the additional services or features are specified in a formal language whilst the properties that the system should exhibit are (typically) specified through the use of temporal logic. A wide range of modelling languages have been used, including Finite State Machines (FSMs), LOTOS, Petri-Nets, Promela and SDL. However, as the number of services grow, there is clearly an issue of the scalability of such techniques and tools. Importantly though, major improvements have been forthcoming in model-checking techniques recently, for example through the use of on-the-fly and symbolic techniques and also the use of abstractions or symmetries. Such techniques can help to greatly reduce the state-space explosion problem. A further problem however is that the level of success is dependent on the accuracy and level of abstraction of the specified properties. An inaccurate (or rather, not precise enough) property specification will inevitably lead to missed interactions (as occurred in [Bousquet99]). Off-line techniques must also rely on a-priori knowledge of the behaviour of the individual services and features.

In contrast, *adaptive on-line* techniques address this latter issue. Such approaches have been developed from a much more pragmatic perspective and have evolved over time to

become increasingly (dynamically) adaptive. Adaptation strategies have typically been powered by a knowledge database, such as predefined tables, state transition rules, abstract data types and user agent rules. For example, in [Griffeth94] unknown new features are accommodated through an adaptive "agent regime" architecture where user agents engage in negotiation to settle the discerned conflicts between features.

Finally, in recognition of the advantages (and certain drawbacks) of both off-line and on-line techniques, a *hybrid* approach is proposed in [Calder99]. This approach is targeted at resolving interactions between new services and legacy services and combines an on-line, transactional approach with off-line formal analysis (see also [Marples00]).

5. Conclusions

It should be apparent from the discussions above that many areas of computer science can benefit from results in the field of feature interaction. We have identified a number of areas where we believe this to be the case including mobile and multimedia systems, component-based and reflective middleware, and also behavioural specification in virtual environments.

An interesting first line of research is to consider the impact of components on the feature interaction problem, including for example the potential role of explicit context dependencies (required/ provided interfaces) [Szyperski98] in simplifying analysis of potential interactions. More generally, further research is clearly required, including strong collaboration between the different research communities, in order to more fully understand the relationships between feature interactions and these other areas.

Acknowledgements

The authors would like to thank a number of researchers at Lancaster who have contributed to discussions on the interactions documented above. In particular, we would like to thank Adrian Friday and Keith Cheverst for their insights into mobile computing, the various members of the Open ORB project for discussions on component-based and reflective middleware (http://www.comp.lancs.ac.uk/computing/research/mpg/reflection/memb.html), and Paul Okanda for his thoughts on behavioural interactions in virtual environments.

References

[Blair98] Blair G.S., Coulson G., Robin P., Papathomas M., "An Architecture for Next Generation Middleware", Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), Springer, 1998.

[Blair00] Blair L. & Pang J., "Feature Interactions - Life Beyond Traditional Telephony", In [Calder00], pp 83-93, 2000.

[Blair01] Blair G.S., Coulson G., Andersen A., Blair L., Clarke M., Costa F., Duran-Limon H., Fitzpatrick T., Johnston L., Moreira R., Parlavantzas N., Saikoski K., "The Design and Implementation of OpenORB v2", To appear in IEEE DS Online, Special Issue on Reflective Middleware, 2001.

[Bousquet99] du Bousquet L., "Feature Interaction Detection using Testing and Model-Checking: Experience Report", In World Congress on Formal Methods, Toulouse, France, Springer, 1999.

[Boussinot01] Boussinot F., Susini J.F., Dang Tran F., Hazard L., "A Reactive Behavior Framework for Dynamic Virtual Worlds", Proceedings of the Web3D 2001 Conference, Paderborn, Germany, February 2001.

[Boussinot00] Boussinot F., Susini J.F., "Junior Rewrite Semantics" Inria Research Report, Available from http://www-sop.inria.fr/meije/rp/junior/Semantics/rewrite-semantics.pdf, October 2000.

[Calder99] Calder M., Magill E., Marples D., “Hybrid Approach to Software Interworking Problems: Managing Interactions between Legacy and Evolving Telecommunications Software”, IEE Proceedings - Software, Vol. 146, No. 3, pp167-175, June 1999.

[Calder00] Calder M., Magill E. (eds), “Feature Interactions in Telecommunications and Software Systems VI”, Glasgow, Scotland, Amsterdam: IOS Press, 2000.

[Cameron94] Cameron E.J., Griffeth N.D., Lin Y.J., Nilson M.E., Schnure W.K., Velthuijsen H., “A Feature Interaction Benchmark for IN and Beyond”, Proceedings of the 2nd International Workshop on Feature Interactions in Telecommunications Systems, Bouma W., Velthuijsen H. (eds), Amsterdam: IOS Press, pp1-23, 1994.

[Dini97] Dini P., Boutaba R., Logrippo L. (eds), “Feature Interactions in Telecommunications Networks IV”, Montreal, Canada, Amsterdam: IOS Press, 1997.

[Efstratiou00] Efstratiou C., Cheverst K., Davies N., Friday A., “Architectural Requirements for the Effective Support of Adaptive Mobile Applications”, work in progress paper in Middleware 2000, New York, April 2000.

[Efstratiou01] Efstratiou C., Cheverst K., Davies N., Friday A., “An Architecture for the Support of Adaptive Context-Aware Applications”, Proceedings of Mobile Data Management (MDM 2001), Hong Kong, January 2001.

[Fireworks97] “FIREworks: Feature Integration in Requirements Engineering”, Esprit Working Group 23531, M. Ryan (Coordinator), started 1997. See http://www.cs.bham.ac.uk/~mdr/fireworks/casestudies.html.

[Griffeth94] Griffeth N.D., Velthuijsen H., “The negotiating agents approach to runtime feature interaction resolution”, Proceedings of the 2nd International Workshop on Feature Interactions in Telecommunications Systems, Bouma W., Velthuijsen H. (eds), Amsterdam: IOS Press, 1994.

[Hall98] Hall R.J., “Feature Combination and Interaction Detection via Foreground/ Background Models”, In [Kimbler98], pp 232-246, 1998.

[Hall00] Hall R.J., “Feature Interactions in Electronic Mail”, In [Calder00], pp 67-82, 2000.

[Heilmeier98] Heilmeier G.H., chairman emeritus of Bellcore (Morristown, N.J.), keynote address at the 35th Design Automation Conference, quote reported in “New telecom services keep vendors on their toes”, Santarini M., http://eetimes.com/dac98/news_telecom.html.

[Kimbler95] Kimbler K., Velthuijsen H., “Feature Interaction Benchmark”, Discussion paper for the panel on Benchmarking at FIW’95 (Feature Interaction Workshop), 1995.

[Kimbler98] Kimbler K., Bouma L.G. (eds), “Feature Interactions in Telecommunications and Software Systems V”, Lund, Sweden, Amsterdam: IOS Press, 1998.

[Marples00] Marples D., “Detection and Resolution of Feature Interactions in Telecommunications Systems During Run-time”, PhD thesis, Available from Dept. of Electrical and Electronic Engineering, University of Strathclyde, Glasgow, August 2000.

[Medvidovic00] Medvidovic N., Taylor R.N., “A Classification and Comparison Framework for Software Architecture Description Languages”, IEEE Transactions on Software Engineering, Vol. 26, No. 1, pp. 70-93, January 2000.

[Plath98] Plath M., Ryan M., “Plug-and-play features”, In [Kimbler98], pp150-164, 1998.

[Shaw96] Shaw M., Clements P., “A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems, Computer Science Department and Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213, April 1996.

[Szyperski98] Szyperski C., “Component Software: Beyond Object-Oriented Programming”, Addison-Wesley, 1998.

A proposal for uniform abstract modeling of feature interactions in UML

Matthias Clauß

Intershop Research
Intershop Communications AG
Jena

Dresden University of Technology
Department of Computer Science
Software Engineering Group

Email: Matthias.Clauss@gmx.de

April 20, 2001

Abstract

The underlying approach presumes the necessity of explicit modeling of variability, partially expressed in features and their interactions. Preceding a UML extension for feature modeling and variability in software models has been developed. A part of the extension deals with the modeling of interactions between features and interactions between the implementation of a feature.

This position paper presents an approach to differ several types of interactions from the view of software modeling. Interactions are distinguished in two abstraction levels related to features and their implementation. The objective is to propose a uniform notation for modeling interactions based on UML that merges feature specification and implementation in a uniform modeling notation.

Keywords: variability, UML notation, classification, feature modeling

1 Background

At the moment no commonly accepted modeling notation exists for the purposes of feature modeling. There are various approaches for feature modeling and the graphical description in feature diagrams, as in [Kang90, Kang98, Griss98]. These approaches typically distinguish three types of features and additional constraints between them. These constraints describe requires- and mutual-exclusion-interactions between features. The feature hierarchy expresses an implicit kind of constraints, but these describe purely selection rules. As typical for the field of feature modeling every approach uses his own modeling notation, partially based on the Unified Modeling Language (UML, [OMG01]).

The UML is a well-known, accepted notation for modeling single software products. It includes an extension mechanism allowing user-defined adaptations for special purposes. But, this standardized extension mechanism is limited to the use of stereotypes and tagged values, completed with constraints on these stereotypes. Other possibilities for extension are notes or changes in the metamodel. Metamodel changes reduce the interchangeability of models and are not supported by modeling tools. Notes can be used but have no relation to the model and therefore lacks of consistency. Both ways are either insufficient or proprietary and should not be used for serious modeling.

For the needs of software family engineering (and software product lines) the author has developed an extension for UML based on the standardized extension mechanisms. It consists of an extension for feature modeling in UML and two smaller extensions to model variability in software products, e.g. during analysis and design of the system. Parts of this extension are the interactions inside the feature model on the one hand and between their implementation on the other hand. This paper focuses on the aspects of feature interaction found useful for modeling purposes.

2 Position

A graphical representation allows a fast and intuitive recognition of arising problems from the modeled interactions. Modeling software is an essential step before implementing a software system and the resulting models are typically used as a common communication base for all participating developers.

One prerequisite for the development of a modeling notation is to know what elements such an extension must contain. Having such an extension a uniform notation could be established for modeling feature interactions.

2.1 Feature interaction in feature modeling

We use the term feature on a high modeling level and define a feature as a recognizable characteristic of a system [Czarnecki00].

Feature modeling with feature diagrams was first introduced in Feature-Oriented Domain Analysis (FODA, [Kang90]). Almost every approach such as FeatuRSEB [Griss98], FORM [Kang98] or Svahnberg et al [Svahnberg] reuses the FODA-concept without major changes. Partially UML is used but always in a non-standardized way. There are used two types of constraints: mutual exclusion and requires. In the context of feature modeling these primarily reflect rules for selection of features but they emerge from feature interactions. Thus these constraints are a way modeling feature interactions.

The use of UML modeling elements introduces a new aspect to these constraints: direction. As dependencies semantically fit best they are used to model feature constraints and therefore constraints can be separated into uni- and bi-directional constraints. This makes sense since a feature can require or exclude another feature, but not vice versa.

These hard constraints are supplemented by weak constraints that should be explicitly modeled if identified. For example if a feature conflicts with another feature in some but not all contexts, this weak form of a ‘mutex’-constraint should be modeled as it is, e.g. a dependency labeled with ‘«weakConstraint» mutex’.

The explicit feature modeling including selection rules and interactions between features is a very necessary step in software modeling for reuse and raises the opportunity to recognize possible problems early.

2.2 Feature interaction in implementation

The underlying approach uses variation points (introduced in [Jacobson97]) to model variability in analysis and design models and consists of the location (the variation point) and recognized implementations (the variants) of variability. To model interactions between variants and between a variant and other model elements constraints are used. These constraints generally describe interactions between the participating elements.

Constraints for variation points are distinguished into three types: mutual exclusion, requires and evolutionary. ‘Requires’ and ‘mutex’ has the same semantics as for feature models but applied on the implementation of features. The need to model evolutionary interactions between model elements has been recognized from [Bosch99]. They could be further separated into, e.g. ‘replaces’, ‘extends’ and ‘decomposes’. This can be used to describe interactions between components for example to model their evolution and explicitly state their dependencies.

These constraints relate with feature interactions in several ways: First variation points can be used to model the implementation of features or feature combinations and thus reflect feature interactions on lower level (analysis, design, maybe implementation). Second, interactions of additionally recognized variation points are explicitly modeled by the constraints. And at last, variation points can be seen as abstractions of variability on code level and even reflect the interactions emerging there.

2.3 Layer-spanning interactions

Besides interactions on each modeling layer, there are layer-spanning relations between modeling elements.

Traceability is needed to understand the development of a concept, e.g. a feature, to one or more implementing elements. In opposite direction, many interactions between features are caused by the implementation. Thus traceability provides a way to trace interactions down to the source and back.

Besides traceability variants consists of a condition specifying the selection rule for the element. This should be expressed in the Object Constraint Language (OCL), a part of the UML. This enables a formal description of conditions and can be used for automatization. As these conditions depend on the context of the described element they are a way to implicitly express interactions that could not be modeled explicitly. In addition, as OCL is a formal language, it enables a formal description of the influence of interactions.

Since it is intended to let these condition refer to the feature modeling layer it is possible to model layer-spanning interactions, e.g. if the use of a variant depends on the existence of several features.

3 Summary

To provide a flexible and broadly applicable notation it is necessary to support a broad but determined range of interactions. To establish a first classification they are separated into several types: mutex, requires and weak-constraints for the feature modeling layer and requires, mutex and evolutionary constraints for implementing modeling layers.

The modeling notation of constraints in UML is completed by the specification of conditions in OCL that enables the detailed and formal description of interactions and context-dependencies.

A broadly accepted classification is needed to get a consistent and generally usable extension that would support modeling of (feature) interactions. This paper drafts a first step towards this.

4 Acknowledgments

The idea of the underpinning diploma thesis originated from discussions with Bogdan Franczyk at the research group of Intershop.

I would like to acknowledge the discussions with Professor Hußmann and Mike Fischer and of course the great support of Bogdan Franczyk who led me to this interesting field of computer science.

References

[OMG01] Object Management Group (OMG), *OMG Unified Modelling Language Specification*, versions 1.3 and 1.4 Draft, March 2000 / February 2001

[Kang90] K. Kang et al, *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, Technical Report No. CMU/SEI-90-TR-2, November 1990, Software Engineering Institute, Carnegie Mellon University, Pittsburgh

[Jacobson97] I. Jacobson, M. Griss, P. Jonsson, *Software Reuse — Architecture, Process and Organization for Business Success*, Addison-Wesley Longman, 1997

[Griss98] M. Griss, J. Favaro, M. d'Alessandro, *Integrating Feature Modelling with the RSEB*, International Conference on Software Reuse, June 1998

[Czarnecki00] K. Czarnecki, U. Eisenecker, *Generative Programming — Methods, Tools and Applications*, Addison-Wesley, 2000

[Kang98] K. Kang, et. al., *FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures*, In Annals of Software Engineering 5, 1998

[Bosch99] J. Bosch, *Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study*, In Proceedings of the 1st Working IFIP Conference on Software Architecture, February 1999

[Svahnberg] M. Svahnberg, J. van Gurp, J. Bosch, *On the notion of Variability in Software Product Lines*

What's in a Name?

Erik Ernst

Presented at the FICS'01 workshop

Abstract

Feature interaction may arise in many different ways, but one of the core topics is the issue of name binding: When two or more entities are composed, say A and B, and they provide more than one declaration of the same name, say n, should the composed entity contain one subentity under that name n, or should it contain several? If one, how should it be selected or constructed? If several, how could a client choose the appropriate one amongst them? This paper surveys the treatment of various problems in this area in several programming languages, thereby establishing a framework for the discussion.

1 Introduction

In many approaches to feature interaction management—including most other papers at this workshop—a software system is assumed to be constructed by certain means which are not of primary interest (the system could be written in some programming language), and the core topic is then to investigate this system by means of formal methods, testing, run-time checks, etc., in order to detect unwanted feature interactions, prove their absence, or similarly.

This paper complements all those techniques in that it focuses on the *construction* of software systems, on making it less likely that a software system is created with unexpected feature interactions in the first place. This is done by focusing on programming languages, and in particular by focusing on a mechanism which plays a crucial role in the emergence of feature interactions. That mechanism is *identification* of declarations, i.e., the rules that govern when two declarations will be considered as describing the same entity.

This implies that we do not consider many important kinds of feature interaction. E.g., one feature of an entity A could be some hard real-time guarantees, and one feature of B could be flexible, customizable support for logging; presumably an entity C created by composing A and B would not satisfy the same real-time requirements as A, because the logging in B could take unbounded time at points where A depends on almost immediate progress. Such kinds of interaction is beyond the scope of this paper. We concentrate on features associated with declared names in programs, so a real-time property would not be a feature. Moreover, feature interaction as considered in this paper will only occur when

two declarations are identified. However, we believe that even this restricted notion of feature and interaction includes a large class of important cases.

The topic of declaration identification is treated in Sect. 2. Section 3 deals with the *effects* of identification, i.e., the meaning of that single entity that is denoted by several declarations which are identified with each other. The last topic, considered in Sect. 4, is *problems* associated with those effects. Finally, Sect. 5 concludes.

2 Identification

The topic of declaration identification is the basis for feature interaction as it is treated in this paper. Given two different declarations of the same name n in some entity, those declarations are said to be *identified* with each other if the meaning of n i context of the entity as a whole must be described by considering the declarations as a group. Alternatively, the two declarations could have unrelated meanings, and there would then have to be some disambiguation rule in order to make both declared meanings of the name n available for clients—in that case the declarations are said to be *distinct*. For example:

```
class A { void f(); virtual void g(); };
class B: public A { void f(); void g(); }
```

Box 1

In box 1, expressed in C++ [9], there are two classes `A` and `B`, and they both contain a declaration of two methods `f` and `g`. Since `B` inherits from `A`, an instance of `B` contains two method implementations called `f` and two method implementations called `g`. However, since `A::g` is declared to be virtual and `A::f` is not, the two declarations of `g` will be identified, and the two declarations of `f` will not.

We should note that identification is traditionally applied to declarations of methods but not to declarations of instance variables. This is probably because instance variables are mutable whereas methods are usually pure values, so instance variables may vary by simply assigning to them whereas methods can only vary if some other mechanism is employed. This makes methods less flexible than instance variables, but it saves a lot of space, it simplifies compile-time checking of the correctness of calling a given method on an instance of a given class, and it improves program comprehension, because the method implementation is known statically for each class.

2.1 Non-Identification

The oldest and simplest approach to declaration identification is to avoid it. In languages such as C [7] and Pascal [5] there are no mechanisms to compose entities such as Pascal `records` or C `structs` and thereby bring declarations together in the same context. Within one declaration block (e.g., the body of a `struct`), declared names must be distinct. Similarly, functions and procedures declared at the same level must have different names.

As a consequence, the question of identification never arises. Moreover, it is sufficient to investigate one declaration in order to learn about the declared entity—whatever holds

according to that declaration will always hold when accessing that entity. In languages with static bindings from name applications to name declarations, this enables excellent run-time performance of procedure calls and attribute access, and that probably made non-identification the default rule for member functions in C++ (like `f` in box 1).

2.2 Identification by Spelling

When identification by spelling is used, two declarations of a name n in one entity are identified with each other because the declared name is n in both cases. So, declarations of the same name in the same context are *always* assumed to denote the same thing.

At first, this rule might seem to be the one that is used in many dynamically typed languages, including Smalltalk [4] and Self [1]. After all, a message '`foo: x`' sent to a Smalltalk object O gives rise to a lookup process (conceptually—it may be implemented in various ways) where the most specific class of O that provides a method with the name `foo:` is allowed to determine the meaning of `foo:` in context of that object.

In Self, a prototype based language that affords programmers an exceptional level of flexibility, the lookup rules are similar to the ones used in Smalltalk. One difference is that each object may have several parents (somewhat similar to having multiple superclasses in class-based languages). The rule applied here requires that there is only one most-specific declaration of the given name, otherwise an "Ambiguous method" error is raised.

It is still the case that the network of classes/objects is inspected, all declarations of `foo:` are considered as a group (conceptually, the implementation may optimize this), and one of them is selected for execution. However, in all these cases we might as well categorize the approach as belonging to the next group, because the number of arguments *are* taken into consideration when matching up method names, as explained in the next section.

2.3 Identification by Spelling and Argument Count

The syntax of languages like Smalltalk and Self ensure that the number of arguments is an integrated part of the name of a method—for instance, `foo` is a method that takes zero arguments, `foo:` takes one argument, and `foo:Bar:` takes two arguments, etc. A message send like `x foo: y Bar: z` invokes `foo:Bar:` on the object `x` with the arguments `y` and `z`. By using that syntactic form (supplemented with another form for 'binary methods') for all message sends, it becomes possible to determine the number of arguments expected by any method, just by looking at its name (its *selector*).

An interesting consequence is that while these languages lack static type checking, they will in fact never call a method with a wrong number of arguments—it would necessarily use a different method selector and thus designate a different method.

In these dynamically typed languages there may be several declarations of methods having the same name. In Smalltalk, there cannot be several instance variables having the same name in the same class, but in Self there is no distinction between instance variables and methods, so one object network may have several instance variables and/or methods with a given name. However, the effect of having multiple instance variables with the same

name is that the most specific one overrides the others; even though this can be used to simulate such features as class variables, it is also somewhat confusing and error-prone.

Finally, a `defmethod` form in a CLOS [6] program introduces a method implementation having a certain name and a certain argument list. This method will become one of the methods 'contained in' some generic function of the same name having a 'congruent' argument list (i.e., primarily, having the same number of required arguments), and when that generic function is called it will select some of the methods and invoke them. In this case the language indisputably identifies methods based on name *and* argument count, because CLOS does not encode the number of arguments into the method name.

Identification by spelling and argument count seems to work well for dynamically typed languages.

2.4 Identification by Signature

In several statically typed languages, including C++ and Java, declarations of methods with the same name may be considered distinct if they take a different number of arguments, or if the types of arguments are (sufficiently) different.

This means that we may have several methods named `foo`, and they are then partitioned into groups according to a combination of the name and the argument types. Within each group the declarations are identified, but declarations in different groups are not.

A common implementation technique used in C++ is to encode the types of arguments into the names of methods, resulting in so-called "mangled" names. Originally, this simplified linking of C++ programs using linkers created for C. Identification by signature may be thought of as identification by name, applied to the mangled names.

It should be noted that this technique of having many meanings of the same name in a given context makes it impossible to use that name in contexts other than the ones where the ambiguity can be resolved. For instance, it would not be possible to extend Java to allow usage of a method in any other way than calling it—e.g., we could not pass a method as an argument to another method—unless some disambiguation scheme were provided.

This approach is generally accepted in connection with statically typed languages even though it has some problems, as pointed out in the next section.

2.5 Identification by Identity

The last approach considered here uses an explicit syntactic representation of each group of identified declarations. It plays the role as the "identity" of that group, and the connection between any given declaration and the identity of the group that it is a member of is resolved at compile-time. Such an approach is taken in `gbeta` [2] and in Lava [8]. For example:

```
A: (# v: < object #);
B: A(# v: :< integer #);
```

Box 2

In box 1, expressed in gbeta, there are two patterns (in this context, think of the word 'pattern' as a synonym for 'class') A and B, both containing a declaration of the name v. B is a subpattern (subclass) of A, so an instance of B will contain two declarations of v.

A declaration in gbeta may consist of a name (here v), then a colon, then a marker that indicates the kind of entity being declared (here '<' or ':<'), and finally the right hand side that specifies the properties of the declared entity.

The marker '<' in A indicates that this declares a pattern, it is virtual, and it is its own identity. The marker ':<' in B indicates that this declares a virtual pattern. It is a requirement that every virtual pattern must have a statically known identity, so the declaration of v in B is only accepted by the static analysis because there is a statically known superpattern (namely A) that provides an identity for this virtual pattern (namely the '<' declaration of v).

The word 'virtual' indicates that the declared entity is specified by means of a group of declarations, and it may be used for extensible methods (similar to virtual member functions in C++ and other OO languages), extensible classes (similar to type arguments for type parameterized classes and methods), and deferred objects (not described here, because of space constraints).

3 Effects of Identification

The most common effect of identification is *selection*, i.e., when the set of identified declarations has been determined, one of them is taken to be the active one and all the others are simply ignored. This is the case with methods in Smalltalk, all kinds of attributes in Self, virtual member functions in C++, and methods in Java. It provides us with the main stream OO semantics of "method overriding".

For a CLOS method, all the member methods in a generic function are taken into consideration. The applicable ones (the ones whose argument constraints are satisfied by the actual arguments) are selected. Amongst the applicable methods, all the methods marked 'before' are executed first, then the most specific ordinary method, then all the methods marked 'after'. Hence, this will *compose* an 'effective' method based on several declarations in the group of identified declarations.

In gbeta the approach is also to compose a resulting meaning out of the meanings of all identified declarations. This happens by means of C3-merging as described in [2, 3].

It should be noted that an approach capable of composing declarations seems to be more suited to work with separation of concerns than an approach based on selection, since the composition may bring together the separated concerns.

4 Problems Caused by Those Effects

Every one of these approaches can give rise to some problems. First, if two methods are conceptually unrelated, they should be able to coexist in one object without disturbing

each other. This works perfectly for non-virtual C++ member functions, but then we cannot redefine them. It is not supported in languages like Smalltalk, Self or CLOS, nor in Java (multiple interfaces with "the same" method must share the implementation of that method in every class implementing those interfaces together). It is supported in gbeta, in that two unrelated methods will have different identities and hence never be identified with each other. This means that a class in gbeta may have two separate methods with the same name *and* still support late binding of both methods. A special problem in this area is the confusion in C++ around identification of methods. For instance:

```
struct A { virtual void f() {}};
struct B: public A { virtual void f() {}};
struct X { virtual void f() {}};
struct Y: public X { virtual void f() {}};
struct BY: public B, public Y {};
struct BY2: public B, public Y { virtual void f () {}};
```

Box 3

There are three groups of declarations of `f` in this example. The groups are {`A::f`, `B::f`}, {`X::f`, `Y::f`}, and {`BY2::f`}. If we execute `f` on an instance of `B` or `BY` then we get `B::f`, both with `A` and `B` as the statically known class. Similarly, if we execute `f` on an instance of `Y` or `BY` then we get `Y::f`. This means that we do get late binding and that the first and second group of declarations above are considered distinct in `BY` that contains both. However, if we execute `f` on an instance of `BY2` then we get `BY2::f` in all cases.

This means that there are two distinct groups of declarations of `f` in all classes except `BY2`, where they are merged into one group. The problem with this is that a programmer who knows about the `A::f` "family" of declarations and who redefines it in `BY2` may accidentally capture the `X::f` family, thus destroying `X::f` for all callers. A programmer who does not know about the existence of `BY2` will have no hint that `A::f` and `X::f` may in certain cases be considered the same method. This problem does not exist when identification by identity is used, because the identities `A::f` and `X::f` would be different.

A similar problem is that a declaration of a method may accidentally override another method of the same name, e.g., if `B::f` were conceptually unrelated to `A::f` but the programmer overlooked the latter. With identification by identity it is syntactically marked out exactly which method declarations are supposed to introduce an entirely new declaration group and which ones are supposed to add a declaration to an existing (and statically known) group. This problem gets even worse in a language like CLOS, where the equivalent of `class BY` would already let `B::f` capture the `X::f` family, again silently.

In CLOS this can be used constructively in 'mixin' classes. However, gbeta demonstrates that mixins can also be used with identification by identity—a bit less flexible, but also safer.

Finally, it is a problem with very strict approaches like identification by identity that methods which are conceptually related but not declared to be so cannot be considered as the same method. However, since problems associated with missing identification generally show up at compile time, we think that it is less dangerous than problems associated with overly aggressive identification, which tend to be undetected until run-time.

5 Conclusion

In this paper, feature interaction was addressed with special attention to the topic of name binding, especially focusing on the subtopic of declaration identification and composition. Different approaches taken in various programming languages were presented, and a few problems associated with each of them mentioned. Generally, we take the position that declaration identification should be statically known, because unforeseen and conceptually unsound identification may cause disasters at run-time, whereas lack of identification that occurs as a compile-time problem for programmers may usually be handled by a bit of extra programming.

References

[1] Ole Agesen, Lars Bak, Craig Chambers, , Bay-Wei Chang, Urs Hölzle, John Maloney, Randall B. Smith, David Ungar, and Mario Wolczko. *The Self 4.0 Programmer's Reference Manual*. Sun Microsystems, Inc., Mountain View, CA, 1995.

[2] Erik Ernst. *gbeta – A Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, DEVISE, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 1999.

[3] Erik Ernst. Propagating class and method combination. In Rachid Guerraoui, editor, *Proceedings ECOOP'99*, LNCS 1628, pages 67–91, Lisboa, Portugal, June 1999. Springer-Verlag.

[4] Adele Goldberg and David Robson. *Smalltalk–80: The Language*. Addison-Wesley, Reading, MA, USA, 1989.

[5] K. Jensen and N. Wirth. *Pascal User Manual and Report*. Springer-Verlag, 1978.

[6] Sonya E. Keene. *Object-Oriented Programming in Common Lisp*. Addison-Wesley, Reading, MA, USA, 1989.

[7] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[8] Günter Kniesel. Type-safe delegation for run-time component adaptation. In *Proceedings ECOOP'99*, LNCS 1628, Lisboa, Portugal, June 1999. Springer-Verlag.

[9] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.

Predicting Feature Interactions in Component-Based Systems

Judith Stafford and Kurt Wallnau
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, USA

+1 412-268-5051 +1 412-268-3265
jas@sei.cmu.edu kcw@sei.cmu.edu

ABSTRACT

Software component technologies support assembly of systems from binary component implementations that may have been created in isolation from one and another. While these technologies provide assistance in wiring components together they fail to provide support for predicting the quality and behavior of configurations of components prior to actual system composition. We believe that all quality attributes manifested at runtime are emergent properties of component interactions, and hence arise as a consequence of planned, or unplanned, interactions among component features. In this paper we discuss the affinities among software architecture, software component technology, compositional reasoning, component property measurement, and component certification for the purpose of mastering component feature interaction, and for developing component technologies that support compositional reasoning, and that guarantee that design-time reasoning assumptions are preserved in deployed component assemblies.

1. Introduction

Software component technologies provide a means for composing systems quickly from precompiled parts. Technologies such as CORBA and COM have been developed to support composition of components that are created in isolation, perhaps by different people in different environments and in different languages. However, current component-based technologies do not support reasoning about system quality attributes, e.g., performance, reliability, and safety.

The quality of a software system is, in part, a function of the degree to which its features interact in predictable ways. Users view systems from the perspective of system features whereas developers view systems in terms of functional decomposition into components. The former is a view in the problem domain; the latter is associated with the solution domain. Turner et al. study the relationship between these two domains as they define a conceptual framework for feature engineering [23]. Quality attributes such as performance, reliability, and safety are emergent properties of patterns of interaction in an assembly of components. Ultimately, all such patterns of interaction depend upon one or more features. Therefore, many critical system quality attributes are expressions of component feature interaction. Indeed, a failure to achieve system quality attributes may be attributable to unexpected feature interaction. We suggest that predicting and ensuring system-level quality attributes and controlling component feature interactions are closely related. Moreover, we contend that the solution to both problems (to the

extent they are distinct) will be found in the form of compositional reasoning. Informally, compositional reasoning posits that if we know something about the properties of two components, *c1* and *c2*, then we can define a reasoning function *f* such that *f(c1, c2)* yields a property of an assembly comprising *c1* and *c2*.

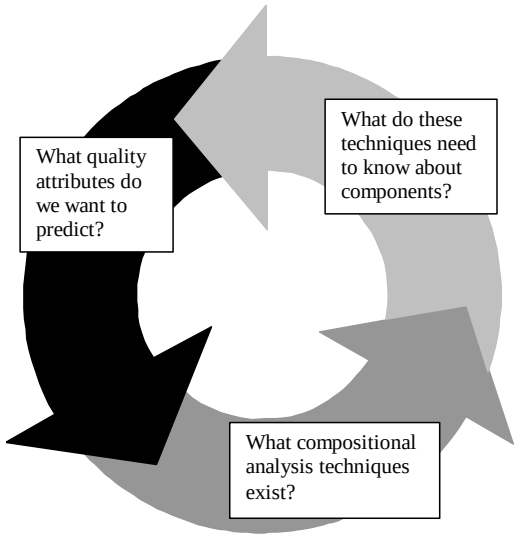
Many would argue that compositional reasoning is the holy grail of software engineering: a noble but ultimately futile quest for an unobtainable objective. This argument usually has as its unspoken premise that only a fully formal and rigorous *f(c1, c2)* will do. If we accept this premise, then progress will indeed be slow. Instead, we suggest that it is possible to adopt a more incremental approach that involves many levels of formality and rigor. To begin, we suggest that three interlocking questions must be answered:

1. What system quality attributes are developers interested in predicting?

2. What analysis techniques exist to support reasoning about these quality attributes, and what component properties do they require?

3. How are these component properties specified, measured, and certified?

Since compositional reasoning ultimately depends upon the types of component properties that can be measured, these questions are interdependent. Therefore, answers to these questions are mutually constraining. Further, answering these questions will be an ongoing process: new prediction models will require new and/or improved component measures, which will in turn lead to more accurate prediction, and to demand for better or additional prediction models.



The objective of our work in predictable assembly from certifiable components (PACC) is to demonstrate how component technology can be extended to support compositional reasoning. To do this, PACC integrates ideas from research in the areas of software architecture, trusted components, and software component technology.

The rest of the paper is organized as follows: We begin by describing two areas of related work, architecture-based analysis and component certification. The former deals with issues antecedent to compositional reasoning, the latter with issues of component trust and specification. We then describe a reference model for using component technology to link compositional reasoning with component certification, and close with a summary of our position.

2. Background and Related Work

The ideas of architectural analysis and component certification are not new but, to the best of our knowledge, their integration is. In this section we describe prior work in these areas and discuss their relationship to our work on predictable assembly.

2.1 Architectural Analysis

Software architecture-based analysis provides a foundation for reasoning about system completeness and correctness early in the development process and at a high level of abstraction. To date, research in the area has focused primarily on the use of architecture description languages (ADLs) as a substrate for analysis algorithms. The analysis algorithms that have been developed for these languages have, in general, focused on correctness properties, such as liveness and safety [2,10,14,16]. However, other types of analysis are also appropriate for use at the architecture level and are currently the focus of research projects. Examples include system understanding [13,21,27], performance analysis [3,20], and architecture-based testing [4,24]. One still unresolved challenge for architecture technology is to bridge the gap between architectural abstractions and implementation. Specification refinement is one approach that seeks to prove properties of the relationship between abstract and more concrete specifications, either across heterogeneous design notations [8] or homogeneous notations [17].

2.2 Component Certification

The National Security Agency (NSA) and the National Institute of Standards and Technology (NIST) used the trusted computer security evaluation criteria (TCSEC), a.k.a. "Orange Book.[1]" as the basis for the *Common Criteria*[2], which defines criteria for certifying security features of components. Their effort was not crowned with success, at least in part because it defined no means of composing criteria (features) across classes of component. The Trusted Components Initiative (TCI)[3] is a loose affiliation of researchers with a shared heritage in formal specification of interfaces. Representative of TCI is the use of pre/post conditions on APIs [15]. This approach does support compositional reasoning, but only about a restricted set of behavioral properties of assemblies. Quality attributes, such as security, performance, availability, and so forth, are beyond the reach of these assertion languages. Voas has defined rigorous mathematical models of component reliability based on statistical approaches to testing [26], but has not defined models of composing reliability measures. Commercial component vendors are not inclined to formally specify their component interfaces, and it is not certain that it would be cost effective for them to do so. Shaw observed that many features of commercial components will be discovered only through use. She proposed component credentials as an open-ended, property-based interface specification [19]. A credential is a triple <attribute, value, knowledge>, which asserts that a component has an attribute of a particular value, and that this value is known through some means. Credentials reflect the need to address component complexity, incomplete knowledge, and levels of confidence (or trust) in what is known about component properties, but do not go beyond notational concepts. Therefore, despite many efforts, fundamental questions remain. What does it mean to trust a component? Still more fundamental: what ends are served by certifying (or developing trust) in these properties?

3. PACC Approach

The PACC approach is based on two fundamental premises: first, that system quality attributes are emergent properties adhere to patterns of interaction among components, and, second, that software component technology provides a means of enforcing predefined and designed interaction patterns, thus

[1] http://www.radium.ncsc.mil/tpep/library/tcsec/index.html

[2] http://csrc.nist.gov/cc/

[3] http://www.trusted-components.org/

facilitating the achievement of system quality attributes by construction.

3.1 Premises of PACC

The study of software architectural styles supports the first premise. An *architectural style* is a recurring design pattern, usually expressed as a set of component types and constraints on their allowable interactions [1,7]. Architectural styles provided the first link between structural design constraints and system properties. For example, the *pipe and filter* style yields systems that can be easily restructured. However, the link between system-level quality attribute and architectural style is informal and subjective. To better formalize this link, Klein et al. have developed *attribute-based architectural style* (ABAS) [11]. Informally, ABAS associates one or more *attribute reasoning frameworks* with an architectural style. An attribute reasoning framework consists of a response variable, one or more stimuli variables, and an analysis model that links stimuli to response. ABAS is a key foundation for PACC. It provides the conceptual foundation for defining and analyzing the properties of assemblies (the response variables). It also provides the link between system properties and component properties (stimuli variables).

Component technology provides the means to realize ABAS concepts in software and, in fact, the concept of architectural style is quite amenable to a component-based interpretation [4]. In or view, a component technology can play an analogous role to predictable assembly that structured programming languages and compilers played for structured programming—it limits the freedom of designers (programmers) so that the resulting design (program) is more readily analyzed. In one of many possible examples, the Enterprise JavaBeans (EJB) specification defines component types, such as *session* and *entity* beans,[4] and constraints on how they interact with one another, with client programs, and with the runtime environment. However serendipitous it may be, it is clear that EJB specifies an architectural style. It is our thesis that analogous component technologies can be defined that go still further to include the additional style constraints needed to support ABAS-based reasoning. The result will be component technologies that support design-time quality attribute analysis, and guarantee, by construction, that the assumptions underlying these analyses are preserved in an assembly of components.

At this point in our research, we are noncommittal about what a prediction-enabled component technology should look like. However, we postulate the outlines of such a technology with the following reference model.

3.2 A Conceptual Reference Model for PACC

Component technologies comprise four levels of abstraction. We generally depict this as a layered reference model, but omit the graphic here for brevity. We describe this model beginning with the concrete and work our way up to the abstract:

– **Assembly**. The most concrete level of our reference model comprises a set of components whose resources (features) have been bound in such a way as to enable their interaction.

– **Assembly specification**. At this level we find component specifications in place of components, and specifications of their interactions. It is at this level of abstraction that attribute analysis and

[4] Components are denoted as *beans* in EJB.

prediction occur.

- **Types**. At this level we specify component and connector types and their features, thereby defining a vocabulary to support design, that is, assembly specification and attribute analysis and prediction.
- **Metatypes**. At this level one defines what it means *to be* a component type, or a connector type, or an assembly type, and define any constraints that must hold for all types to enable attribute prediction.

3.3 Reference Model Instantiations

We have explored two complementary approaches to instantiate the PACC reference model: one that assumes that attribute reasoning models will be integrated into a component technology, and one that assumes the converse. We refer to the first as a component-centric instantiation, and the second as an architecture-centric instantiation. We have validated both approaches with (admittedly simple) proofs of feasibility. For the component-centric instantiation we used the WaterBeans [18] technology augmented with latency prediction. For the architecture-centric instantiation we used a security ABAS for attribute reasoning, and a Web-based enterprise system for the component technology (from the case study found in [25]). Table 1 summarizes the mapping of these instantiations to the reference model.

Table 1: Complementary Instantiations

Model Level	Component Centric	Architecture Centric
Metatypes	Properties shared by all WaterBeans components, e.g., typed ports, connectors, and connection rules. Defined the latency attribute and associated it with the component metatype.	A simple, behavior-less ADL of components, interactions, assemblies, and their properties. Analogous to a simplified metamodel of UML collaboration diagrams.
Types	Component type definitions for CD audio sampling and wave manipulation. Types introduced the additional Boolean property for aperiodic or periodic behavior, and, if periodic, the execution period. A quantitative model for end-to-end latency is also defined here.	Types that represent basic-level categories for analysis of security properties, e.g., peers, trusted computing base, key, cryptographic provider, threat agent, data asset. Each category is mapped to an element in the simple ADL.
Specification	A topology of audio components annotated with their latency attributes; assembly latency prediction occurred here.	Patterns of interaction comprising only basic categories, where patterns exhibit desired security property. Informal rules of attribute preserving pattern refinement.
Assembly	A benchmarked assembly, allowing comparison of predicted versus actual assembly la-	Pattern refinements where each basic category has been refined to (bound to) a more

	tency.	specific category, ultimately grounding in specific component and interaction features.

4. Closing Thoughts

In closing, we take the position that the identification of feature interactions in complex systems is closely tied to analysis of system-level quality attributes. Quality attributes of systems are a product of properties associated with both the components that comprise a system and their patterns of interaction. Designing systems as assemblies of components based on architectural styles produces systems that are analyzable by design. We are exploring the application compositional reasoning techniques to assemblies of components in order to predict properties of systems. It is our belief that this line of work can support the identification of the potential for feature interaction before actual system assembly.

5. Acknowledgements

This work was supported by the United States Department of Defense.

6. References

1. G. D. Abowd, R. Allen and D. Garlan, Formalizing Style to Understand Descriptions of Software Architecture, *ACM Transactions on Software Engineering and Methodology*, Vol. 4, No. 4, October, 1995, pp. 319-364.
2. R. Allen and D. Garlan, A Formal Basis for Architectural Connection, *ACM Transactions on Software Engineering and Methodology*, Vol. 6, No. 3, July. 1997, pp. 213-249.
3. S. Balsamo, P. Inverardi and C. Mangano, An Approach to Performance Evaluation of Software Architectures, *Proceedings of the 1998 Workshop on Software and Performance*, October. 1998, pp. 77-84.
4. F. Bachman, L. Bass, C. Buhman, S.Comella-Dorda, F. Long, J. Robert, R. Seacord and K. Wallnau, Volume II: Technical Concepts of Component-Based Software Engineering, Technical Report CMU/SEI-2000-TR-08, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.
5. A. Bertolino, P. Inverardi, H. Muccini and A. Rosetti, An Approach to Integration Testing Based on Architectural Descriptions, *Proceedings of the 1997 International Conference on Engineering of Complex Computer Systems*, September. 1997, pp. 77-84.
6. E. Dijkstra, Structured Programming, *Software Engineering, Concepts and Techniques*, J. Buxton et al. (eds.), Van Nostrand Reinhold, 1976.
7. D. Garlan and M. Shaw , An Introduction to Software Architecture, *Advances in Software Engineering and Knowledge Engineering*, V. Ambriola and G. Tortora (eds.), World Scientific, 1993.
8. F. Gilham, R. Reimenschneider, V. Stavridou, Secure Interoperation of Secure Distributed Databases: An Architecture Verification Case Study, *Proceedings of World Congress on Formal Methods (FM'99)*, Vol. I, LNCS 1708, pp. 701-717, 1999, Springer-Verlag, Berlin.
9. G. T. Heineman and W.T. Councill (eds.), *Component-Based Software Engineering: Putting the Pieces Together*, Addison-Wesley, Reading, Massachusetts, 2001.
10. P. Inverardi, A.L. Wolf, and D. Yankelevich, Static Checking of System Behaviors Using Derived Compo-

nent Assumptions, *ACM Transaction on Software Engineering and Methodology*, Vol. 9, No. 3, July. 2000, pp. 238-272.

11. M. Klein and R. Kazman, *Attribute-Based Architectural Styles*, Technical Report CMU/SEI-99-TR-022, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.
12. M. Klein, T. Ralya, B. Pollak, R. Obenza and M. G. Harbour, *A Practitioner's Handbook for Real-Time Analysis*, Kluwer Academic Publishers, 1993.
13. J. Kramer and J. Magee, Analysing Dynamic Change in Software Architectures: A Case Study, *Proceedings of the 4th International Conference on Configurable Distributed Systems*, May 1998, pp. 91-100.
14. J. Magee, J. Kramer, and D. Giannakopoulou, Analysing the Behaviour of Distributed Software Architectures: A Case Study, *Proceedings of the 5th IEEE Workshop on Future Trends of Distributed Computing Systems*, October. 1997, pp. 240-247.
15. B. Meyer, *Object-Oriented Software Construction, Second Edition*, Prentice Hall, London, 1997.
16. G. Naumovich, G.S. Avrunin, L.A. Clarke, and L.J. Osterweil, Applying Static Analysis to Software Architectures, *Proceedings of the 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT Symposium on Foundations of Software Engineering* Lecture Notes in Computer Science, No. 1301, Springer-Verlag, 1997, pp. 77-93.
17. J. Phillips and B. Rumpe, Refinement of Information Flow Architectures, *Proceedings of the 1st IEEE International Conference on Formal Engineering Models*, pp. 203-212, 1997.
18. D. Plakosh, D. Smith and K. Wallnau, *Builder's Guide for WaterBeans Components*, Technical Report CMU/SEI-99-TR-024, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.
19. M. Shaw, Truth vs Knowledge: The Difference Between What a Component Does and What We Know It Does, *Proceedings of the 8th International Workshop on Software Specification and Design*, March 1996.
20. B. Spitznagel, D. Garlan, Architecture-Based Performance Analysis, *Proceedings of the 1998 Conference on Software Engineering and Knowledge Engineering*, San Francisco, California, 1998.
21. J.A. Stafford and A.L. Wolf, Architecture-Level Dependence Analysis in Support of Software Maintenance, *Proceedings of the Third International Workshop on Software Architecture*, November. 1998, pp. 129-132.
22. C. Szyperski, *Component Software Beyond Object-Oriented Programming*, Addison-Wesley, Boston, Massachusetts and ACM Press, 1998.
23. C. R. Turner, A. Fuggetta, L. Lavazza, and A. L. Wolf, A Conceptual Basis for Feature Engineering, *Journal of Systems and Software*, Vol. 49, No. 1, December 1999, pp. 3-15.
24. M.E.R. Vieira , S. Dias and D.J. Richardson, Analyzing Software Architectures with Argus-I, *Proceedings of the 2000 International Conference on Software Engineering*, June 2000, pp. 758-761.
25. K. Wallnau, S. Hissam and R. Seacord, *Building Systems from Commercial Components*, Addison Wesley Longman, To Appear July, 2001.
26. J. Voas and J. Payne, Dependability Certification of Software Components, *Journal of Systems and Software*, No. 52, 2000, pg. 165-172.
27. J. Zhao, Using Dependence Analysis to Support Software Architecture Understanding, *New Technologies on Computer Software*, September 1997, pp. 135-142.

An Architectural Style to Integrate Components and Aspects[1]

Miguel A. Pérez, Amparo Navasa, Juan M. Murillo
Area de Lenguajes y Sistemas Informáticos
Departamento de Informática. Extremadura Univ.
{toledano, amparonm,juanmamu}@unex.es

In the last few years Component Based Development (CBD) and Aspect Oriented Programming (AOP) are gaining more and more relevance. Whilst CBD has been shown as a good mean to reuse designs and to build complex systems by the plug-and-play mechanisms, AOP makes code more reusable and complex systems more comprehensible. However, little efforts have been made to integrate both paradigms and consequently, developers are condemned to don't get all the advantages offered by the two paradigms simultaneously. In this paper a proposal to integrate AOP and CBD is presented. The work introduce the concept of Aspect Component and it is split in two different parts: first, an architectural style to support both, functional and aspect components is proposed and second, an approach to document, search and find aspect components in repositories is presented.

1 Introduction

In the last few years, AOP and CBD have become more relevant, and have showed their use in developing complex systems. Nevertheless, both paradigms have evolved in separate ways:

1. AOP has been shown as a good mean to develop complex systems. This paradigm allows separating the aspect code that contaminates the functional code of applications. Some of the main benefits obtained are to have both, functional and aspects code, more reusable and make the systems more comprehensible. As a consequence the software quality is improved.
2. On the other hand CDB is shown as a powerful paradigm that favours the designs' reuse, using the plug and play mechanism to build software systems. The systems that are built in this way evolve in a simple fashion and are easily maintained. However, in this paradigm, the aspect code is crosscutting the functional code in components.

Although AOP and CBD are conceived to develop complex Software System, this paradigms can not be used both together. The reason is that the actual CBD models do not support aspect separation. This means the developers must trade off one paradigm for the other, and means that they are unable to obtain benefits from both paradigms simultaneously. Nevertheless, intuitively, it seems possible that both paradigms can be joined together successfully [ScAs98]: aspects could be separated from components to be modelled as a special kind of component. This would allow giving the benefits of components to aspects and the benefits of separation of concerns to components.

In this paper, a proposal to combine AOP and CBD is presented. The primary objective is to construct systems based on components that separately treat several (functional and non-functional) concerns. To do this, the concept of *aspect component* is introduced and defined in the following way:

[1] This work has been supported by CICYT, project TIC 99-1083-C2-02

An "aspect component" is a component that has a code associated with a concern (and only one), functional or non-functional.

The contribution of the paper is two-fold: on one hand, it defines an architectural style in the sense [GaSh96] for the construction of component oriented systems, considering non-functional aspects. On the other hand, the work assumes the existence of repositories of aspect components. Retrieving the information from them presents different problems than those already know for the functional component repositories. For these reason it is introduced a model for the documentation, selection and retrieval of aspect components. The model is based on the analysis of the information of interoperability needed for this special kind of components.

2. Proposed model

The first contribution of this paper consists of defining an architectural style for the definition of components oriented complex System. It considers the non-functional aspects that are applied to functional components. In this model, we consider the following definition of a component give by Krunchen in the CBSE workshop in 1998 [Kr98]:

"A component is a non-trivial, nearly independent, and replaceable part of a system that fulfils a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realisation of a set of interfaces."

The software architecture is based on the layered architectural style proposed by Garlan and Shaw [ShGa96]. In the architectural style proposed there is one level for each considered concern (figure1). Each level controls the actions performed in the inferior level according to the requirements of the concern that is implementing. Also, each level is intended to support the plug in of the adequate aspect components.

The proposed model is based on the following principles:

(1) The components that form part of the system have a generic structure. Never they will be considered built by or for the model. Thus, one can suppose that they can be found in public repositories.
(2) Each component has an interface to present its own services.
(3) *Each component needs to be associated with information of interoperability in order to locate them in the repositories and know if their requirements coincide with the sought requirements.*

Figure 1 represents the proposal graphically. It indicates the order in which the aspects should be considered. Note that, whilst the second and third levels are only applied to one component, next levels are applied to several ones. The reason is that co-ordination and distribution concerns relate several components.

Levels in proposed architectural style are:

Level 1: The one called "functional components level" is the lowest. It contains the functional components. Each one has one or more interfaces to define its interaction with the others components in the system.

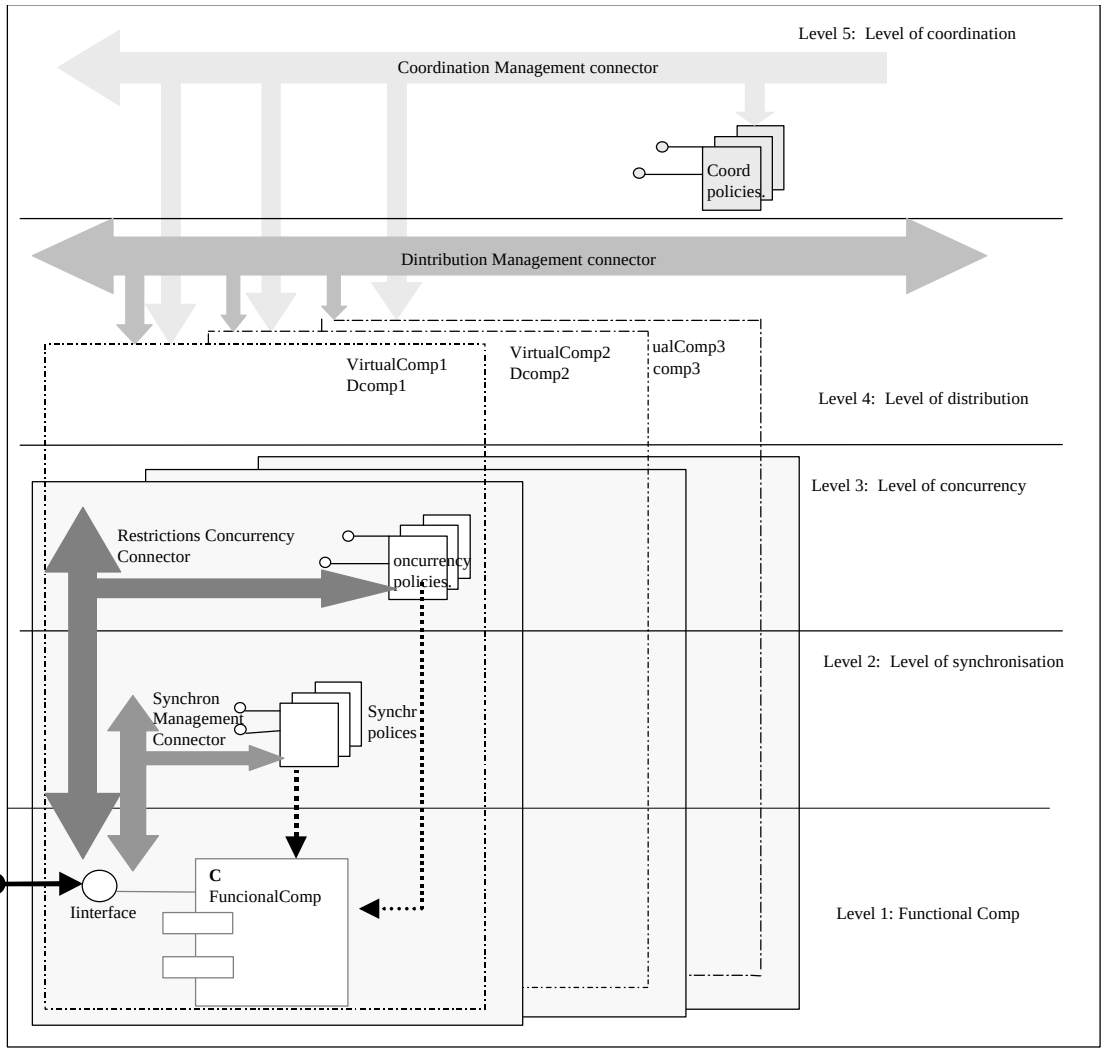


Figure 1. Layered architecture elements.

Level 2: Synchronisation level. This level will decide when, the actions performed by the functional component can be executed. This is made according to the constraints specified by the aspect components plugged in at this level.

Level 3: Concurrency Level. In this level, the concurrency control for components in level 1 is made. Components in this level include constraints specifying how the actions performed by the functional components can be executed concurrently.

Level 4: Distribution level. Using a platform (CORBA, RMI,…) it is possible to access remote components. Platforms manage the components location and their communication. This level is defined over components in previous levels. It defines the distribution policy of components in the system. The model presented here doesn't wish acquire obligations with any platform, but considers the possibility to use several at one time.

Level 5: The co-ordination level implements the dependencies among the actions performed in the whole system. Components at this level specify the co-ordination constraints to apply over the system components (distributed or not).

Each level controls the action performed in the inferior level through a reflective schema: When a service is required to a functional component the requirement is intercepted and processed by the superior levels.

This architecture allows building systems based on functional and non-functional components. However, it is necessary note that components will reside in repositories and that they should have a good interoperability specification for making correct matches when searching them. Next section study this problem.

3 Component selection

The second part of the work presented here consists of searching for and selecting aspects components from repositories. If it is well known the difficulty to retrieve functional components in repositories [HeVaTr00, VaHeTr99], the retrieval of the aspect components is even more complicated. The reasons are the following:

- Aspect components can be developed for different models of separation of concerns [Berg94, Kic96, Lie96, Cza98, Osh96]. Each one of these models is different from the others, which complicates the recuperation. It is needed model independent documentation of interoperability, that is: the documentation of interoperability must be valid for all the designed components for the same aspect (independently of the model that the designer choice to achieve the separation).
- The architectural style proposed collects various aspects, and each aspect needs different interoperability documentation. For example, the semantic description of a component that specifies the co-ordination aspect is different from one specifying the distribution aspect. This is due to the fact that the documentation requirements change when describing each behaviour.
- While the interface of the functional components serves to show the services that this particular component offers, in the aspect components the interface describes the requirements required for the components to which they will be applied.
- Besides, to select a functional component from a repository we need key words, names of components, domains, granularity,... Nevertheless, to select aspect components, apart from those mentioned above, we need to access components in relation to their behaviour, and this is difficult to describe.

Whilst system's construction using functional components [ScAs98] has already been handled by methodologies like Catalysis [SoWi00], composition using aspect components lack methodologies**,** and there are not any public aspect repositories, nor tools to select them.

Our work considers that there exists aspect repositories from which to select the desired components. We have found that to select aspect components, it is needed documentation that includes the following:

1. The documentation of interoperability of the component should contain syntactic information that describes the aspect component's interface. This description could be one that explains to the other components *how they should use* the aspect component.

2. Semantic information that explains the aspect component's behaviour. This information is more difficult to obtain, and requires the use of specific tools, since the needs of documentation change for every aspect. The objective is to generate information about *what the component does.*
3. *General information*, about the model of the component, the name of the component, the most frequently used domain, the size,… This information can be completed with information about the transitions of the component's state to facilitate its comprehension. The idea is to obtain complementary information, which permits a more complete knowledge of the desired component and facilitates its interoperability in different application domains.

The information in point 1 is similar to the syntactic information necessary to use any functional component. But now, this information does not exhibit the services, but rather the requirements, which the components that are used should comply. The information obtained in point 3 is not different from that which a functional component could have. However, the semantic information in point 2 presents new challenges.

To describe semantic information, one needs a tool to describe the behaviour of the different aspect components, permit the selection of components from a repository and compare the behaviour of different components in order to choose them correctly.

Our research began with a study of the co-ordination components. Thus, to document the components [Kr98] that collect the co-ordination aspect, we use diagrams similar to sequence diagrams proposed in UML. However, we have amplified its syntax to cover the needs of the aspect of co-ordination which do not collect the diagrams of sequence [In00, HaPo98], such as representing the notification of event or differentiating control method of the notification of events.

Once we have obtained the diagram(s) that describe the function and interaction with other components, the tool constructs (based on it) the automaton that describes its behaviour. Finally, this automaton is optimised. The obtained automata will be used to search, compare, and select elements from repositories.

Our tool is presented in a graphic environment to collect, in a simple fashion, the entire user's needs: the sequence of evens that the component might receive, the type, the form of notification… From these needs automata are built that are compared with those represented in the repository.

The following steps are for the creation of new tools that permit us to work with other different aspects. The final goal is to have an adequate environment to select any type of aspect components.

4 Conclusions.

In this paper, a software architecture for building complex systems integrating concepts of AOP and CBD has been presented. Also, it has been presented a proposal for the documentation and selection of aspect components from repositories.

Future works that we have proposed are directed toward a formal modellioing of architecture trough an Architectural Definition Language and the creation of tools for the selection and retrieval of aspect components from repositories.

5 Bibliography

[Berg94] L. Bergmans. “Composing Concurrent Object”. Tesis Doctoral. University of Twenty . Netherland. 1994.

[Cza98] K. Czarnecki. Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configurations and Fragment-Based Component Models. Tesis doctoral, Technische Universität Ilmenau, Alemania. 1998.

[HaPo98] D. Harel, M. Politi. “Modeling reactive Systems with Statecharts. The Stamate approach.”Mc Graw-Hills 1998.

[HeVaTr00] J. Hernández, A. Vallecillo, J. M. Troya, “New Issues in Object Interoperability” Proc of the ECOOP´00. Workshop on Object Interoperabiliy. Sophia Anthipolis- France.

[In00] Ingolf Heiko Krüger. “Distributed System Design with Message Sequence Charts”. PhD Thesis. University of, July 2000.

[Kic96] G. Kiczales et al. Aspect-Oriented Programming. In Max Mühlhäuser editor, Special Issues in Object-Oriented Programming, Workshop Reader of the 10th. European Conference on Object-Oriented Programming, ECOOP’96, Dpunkt-Verlag, 1997.

[Kr98] Ph. Kruchten. “Modeling Component Systems with the Unified Modeling Languaje”. International Workshop on CBSE 1998.

[Lie96] Karl Lieberherr . Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns. PWS Publishing Company. 0-534- 94602-X. 1996.

[Osh96] H. Ossher, M. Kaplan, A. Katz, W. Harrison, V. Kruskal. Specifying Subject-Oriented Composition. Theory and Practice of Object Systems, volume 2, number 3, Wiley & Sons. 1996.

[ScAs98] R. Schmidt, U. Assmann. “Conceps for Developing Component-Based Systems”. International Workshop on CBSE 1998.

[ShGa96] M. Shaw, D. Garlan. “Software Architecture: Perspectives on an Emerging Discipline”. Prentice Hall 1996.

[SoWi00] Francis D´Souza, Cameron Wills, “Objects, Componentes and Frameworks with UML. The Catalysis Approach”

[VaHeTr99] A. Vallecillo, J. Hernández, J. M. Troya, “New Issues in Object Interoperability” Proc of the ECOOP´99. Workshop on Object Interoperabiliy. Lisbon, Portugal.

Feature Modelling and Composition with Coordination Contracts

Luís Filipe Andrade[1] and José Luiz Fiadeiro[1,2]

[1] ATX Software S.A.
Alameda António Sérgio 7 – 1 A,
2795 Linda-a-Velha, Portugal
landrade@oblog.pt

[2] LabMAC & Dept. of Informatics
Faculty of Sciences, University of Lisbon
Campo Grande, 1700 Lisboa, Portugal
jose@fiadeiro.org

1 Introduction and motivation

In this paper, we present an approach to feature modelling and composition that relies on the use of what we call "coordination technologies" – a set of modelling primitives, design principles, design patterns, and analysis techniques that we have been developing for supporting the construction and evolution of complex software systems that need to operate in very volatile and dynamic environments. These technologies are based on the separation between what in systems is concerned with the computations that are responsible for the functionality of the services that they offer and the mechanisms that coordinate the way components interact, a paradigm that has been developed in the context of so-called Coordination Languages and Models [15].

The rationale of the application of coordination technologies to Feature Modelling and Composition is in the realization that the problems that arise in this field, namely the so-called "feature interaction problem", can be better addressed in frameworks where this separation between computation and coordination is supported and can be used for representing the ways in which features are composed as first-class citizens. More precisely, our standpoint and justification for the approach to these problems that we are going to present can be summarised as follows:

Feature interaction is a "feature" intrinsic to the nature of complex systems and the approaches that we follow to model them. We are in total agreement with Pamela Zave [23] in recognising that feature-oriented system descriptions make feature interaction, in the sense of the emergence of "strange", "unexpected" or "undesired" behaviour from feature composition, implicit in the models we provide. The whole idea of feature-oriented approaches, in the sense of putting together systems from individual features as basic units of functionality, is that the global properties that are required from the behaviour of the system can "emerge" from the individual functionalities and the interactions that exist between them. With no interaction there is no emergence of new behaviour and, therefore, no value to the system as a whole that is not already provided through its components in isolation. The problem is that, while we compose features having in mind the emergence of certain properties that constitute requirements on the behaviour of the system, it is difficult to predict which other forms of behaviour will also emerge, namely ones that are not of interest and whose "negation" is normally omitted from the requirements

specification because one never thought of them being possible… Hence, feature interaction is not a problem that needs to be solved but a phenomenon that needs to be controlled.

The problem of controlling feature interaction is aggravated by evolution. As the world of business in general becomes more and more aggressive and competitive, for instance as a consequence of the impact of the Internet and Wireless Technologies, companies require their information systems to be easily adaptable to changes in the business rules with which they operate, most of the time in a way that does not imply interruptions to the services that they provide. Quoting directly from [13], "… the ability to change is now more important than the ability to create e-commerce systems in the first place. Change becomes a first-class design goal and requires business and technology architecture whose components can be added, modified, replaced and reconfigured". All this means that the "complexity" of software has definitely shifted from *construction* to *evolution*, and that methods and technologies are required that address this new level of complexity and adaptability. One of the important goals of feature-oriented approaches is, precisely, to make it easier to change system's behaviour by adding new features, changing existing ones, or rearranging the way in which they are composed. All these operations have in mind the emergence of new system properties corresponding to the need to accommodate new business requirements, which brings us back to the problem of making sure that we are not surprised with the emergence of "funny" phenomena. Hence, even if a system was carefully constructed in order to avoid undesired interactions, it is impossible to predict how it is going to evolve and, hence, guarantee that it will be forever free of "bad" interactions. What is even worse is that our own perception of what is "good" and "bad" will evolve according to the changes that occur in the application domain and that prompt the need for the system to evolve as well … Hence, again, we need ways of controlling the evolution of systems that make it easier to reconfigure them dynamically so that interactions can be revised on the spot to correct "harmful" interference between the features in place.

Central to the problem is the way features are composed. This is our main standpoint and the justification for the use of coordination technologies in feature-oriented approaches. More precisely, there are two important aspects that need to be addressed with respect to composition. On the one hand, we need to provide ways for detecting emergent properties from the way individual properties of features and the way they are composed. On the other hand, we need to provide the means for the ways features are interconnected to be given explicit representations in system models so that they can be designed, deployed and evolved without having to intrude in the way the features to which they apply have been deployed. That is to say, our proposal is for the composition mechanisms that apply to features in a system to be externalised and addressed directly as first-class citizens so that they can be used to control interactions within the system without having to change the features.

2 Externalising feature composition through contracts

Our approach to feature modelling and composition is based on the belief that forms of feature composition should be seen as connectors that one may *superpose*, in a non-intrusive way, on the features of systems. The term "superposition" refers to a mechanism, also called "superimposition" by some authors, that has been developed and applied in Parallel Program Design for extending systems while preserving their properties [6,8,14,17]. Our belief is that this basic mechanism is lacking in feature-oriented development approaches and fills the gap that we have identified for supporting the externalisation of interactions and controlling their evolution.

Coordination contracts are the semantic primitives that are at the core of our proposal. A coordination contract fulfils a role similar to that of a *connector* in the terminology of software architectures [1]. It consists of a prescription of *coordination effects* (the *glue* of the connector) that will be superposed on a collection of partners (system features) when the occurrence of one of the contract *triggers* is detected in the system. In the description of the contract, the partners are not identified as specific features of a specific system but in terms of a number of *coordination interfaces* (the *roles* of the connector) that act as types that can be instantiated with features of the system when the contract is activated on a particular configuration. This makes contracts generic feature composition operators that can be independently deployed and reused in different contexts.

The trigger/reaction mode of coordination that we have in mind requires that each coordination interface identifies which events produced during system execution are required to be detected as triggers for the contract to react, and which services must be made available for the reaction to superpose the required effects. The nature of triggers and services can vary depending on the nature of the language, or class of languages, in which the features to which the contract will be applied are, or are likely to be, implemented. In an object-oriented environment, typical events that constitute triggers are calls for operations/methods of instance objects, and typical services are the operations that are made public by the object class. In such cases, coordination interfaces can be identified with abstractions already made available in object-oriented programming through mechanisms such as class interfaces in Java, pure virtual classes in C++, and abstract classes in Eiffel. Another class of events that we have found useful as triggers is the observation of changes taking place in the system. For such changes to be detected, components that implement features must make available methods through which the required observations can be made (something that the mechanism of role instantiation must check), and a detection mechanism must be made available in the implementation platform to enable such changes to be effectively monitored (something that is not universally provided).

When a coordination interface is instantiated with a specific feature as part of the activation of the contract over a given configuration of the system, the interface must be formally related with what the component that implements the feature makes available through its own interface. Again, this process of instantiation will depend on the nature of the deployment of the component itself. For instance, programming languages such as Java already provide mechanisms for interfaces to be implemented through object classes. Ideally, the component definition language should support the distinction between *event* (to be used for triggers) and *services* (to be used for reactions), but this is not necessary. We decided to separate concerns in coordination

interfaces as much as possible as a means of setting the direction in which we think Interface Definition Languages could evolve, but this separation does not need to be enforced for our techniques to be applicable. Hence, for instance, typical notions of interface in which components declare which methods are public can be used: events can be detected as calls to the public methods of the component and services can be effected through the invocation of these methods by the contract.

This separation between the coordination interface and the components that implement the features themselves is an essential mechanism for being able to interconnect heterogeneous features and, hence, not to compromise the ability of the system to evolve by upgrading the way its features are deployed or integrating third-party features in a controllable way.

. A coordination contract is defined as follows:

```
contract <name>
partners <coordination interfaces that instances need to exhibit>
constraints <an invariant that the partners need to satisfy>
attributes
operations
coordination
end class
```

Coordination is prescribed under through a number of rules of the form:

```
<name>     when <trigger>
           with <condition>
           do <set of services>
```

The name of the rule identifies a particular form of coordination; it identifies a point of “rendez-vous” in which the partner instances have to synchronize their behaviour. The names themselves are used for managing the interference between different contracts that may be active in the same state as discussed further below.

For each rule, the condition under “when” identifies the trigger that prompts the contract to become active and coordinate the behaviour of the partner features. Several trigger conditions can be placed in the “when” clause using the keyword “AND”. If one of such conditions is not satisfied, the contract is considered as being “inactive” and, as a result, the participants progress independently of the reaction specified in the rule. This mechanism provides the ability for controlling which of the contracts imposed on a feature will be responsible for coordinating it, thus allowing for dynamic configuration of the behaviour of the feature.

The “do” part of each rule identifies a synchronization set of services of the partners and some of the contracts own actions. This set is required to be executed atomically in the sense that if the execution of any of its actions fails, the execution of the rule itself fails. In [16], we have used a special notation for reactions to be performed on triggers that consist of calls on services, e.g. as in the case of “big-credits”. In such cases, we allow for the synchronisation set to be structured in terms of operations that should be performed *before* the operation called in the trigger and those that should be performed *after*. Another useful feature considered in [16] is to provide a *replace* clause through which a new implementation can be given for the operation that is being called in the trigger, for instance in order to optimise performance or, simply, as a means of upgrading legacy code. See also [7] for suggestions along these lines. In order to preserve semantics, any such replacement is required to satisfy whatever specification has been given for the original operation, namely any contracts in the sense of [21] through pre and post conditions.

Each synchronisation set is guarded by the conjunction of the guards of the individual services together with the conditions included in the "with" clause. Therefore, the “with” clause puts further constraints on the execution of the actions involved in the interaction. If any condition under the “with” clause is not satisfied, an exception is thrown as a result and none of the actions in the synchronisation set is executed. A more detailed description of coordination contracts and the technology that puts them in practice can be found in [2,4,16]. Due to lack of space, it is impossible to give an example with any meaningful content. Hence, we prefer to refer the readers to [24] where several example from financial services can be found, to [18] for telecommunication services, and [19] for stock trading services.

Finally, it is important to mention that in [16] we have shown that, even if none of the standards for component-based software development that have emerged in the last few years (e.g. CORBA, EJB and COM) can provide a convenient and abstract way of supporting the proposed coordination principles as first-class mechanisms, an implementation can be given that is based on a design pattern that exploits some widely available properties of object-oriented programming languages such as polymorphism and subtyping. This pattern supports the forms of compositional, “black box” view of evolution that we motivated in section 1 and that allows us to perform changes on contracts without being intrusive on the components that implement the features under coordination.

3 Semantics and analysis

Besides the need for modelling primitives that allow us to externalise form s of feature composition as first-class citizens, following the principle of “exoskeletal software” [20], we mentioned in section 1 that we need means for detecting emergent properties. This is needed not only to be able to detect “bad” interactions but also to ensure that the “good” ones, i.e. those that enforce the requirements on the global behaviour of the system, are effectively there. For that purpose, we need a formal semantics for our coordination technologies.

The formal semantics that we have developed is based on a categorical framework in which we have formalised notions of coordination [12], namely in the context of architectural languages [11] and parallel program design [10]. The presentation of this semantics is out of the scope of this position paper. In [3], we have shown how the categorical framework relates directly to the evolutionary aspects of coordination contracts, and in [5] we have related it to the design pattern that we provide for contracts.

An analysis of emergence in this categorical framework can also be found in [9]. This analysis relies on the use of logical formalisms for modelling requirements specifications and captures emergence through non-conservative interpretations between theories. This framework will provide the basis for the definition of the techniques that we intend to provide for assisting in the detection of “bad” interactions, but this is a matter of further research. Another important way of assisting in this detection is through the use of animation techniques. These are being deployed in the environments that we provide for contract development [16].

Finally, we should mention that the categorical framework also provides the basis for the language that we are defining for assisting in the (re)configuration of systems. Preliminary results can be found in [22].

References

1. R.Allen and D.Garlan, "A Formal Basis for Architectural Connectors", *ACM TOSEM*, 6(3), 1997, 213-249.
2. L.F.Andrade and J.L.Fiadeiro, "Interconnecting Objects via Contracts", in *UML'99 – Beyond the Standard*, R.France and B.Rumpe (eds), LNCS 1723, Springer Verlag 1999, 566-583.
3. L.F.Andrade and J.L.Fiadeiro, "Coordination: the Evolutionary Dimension", in *Technology of Object-Oriented Languages and Systems – TOOLS 38*, W.Pree (ed), IEEE Computer Society Press 2001, 136-147.
4. L.F.Andrade and J.L.Fiadeiro, "Coordination Technologies for Managing Information System Evolution", in *Proc. CAISE'01*, A.Geppert (ed), LNCS, Springer-Verlag 2001, in print.
5. L.F.Andrade, J.L.Fiadeiro, J.Gouveia, A.Lopes and M.Wermelinger, "Patterns for Coordination", in *COORDINATION'00*, G.Catalin-Roman and A.Porto (eds), LNCS 1906, Springer-Verlag 2000, 317-322
6. R.Back and R.Kurki-Suonio, "Distributed Cooperation with Action Systems", *ACM TOPLAS* 10(4), 1988, 513-554.
7. J.Bosch, "Superimposition: A Component Adaptation Technique", *Information and Software Technology* 1999.
8. K.Chandy and J.Misra, *Parallel Program Design*, Addison-Wesley 1988.
9. J.L.Fiadeiro, "On the Emergence of Properties in Component-Based Systems", in *Algebraic Methodology and Software Technology (AMAST'96)*, M.Wirsing and M.Nivat (eds), LNCS 110143, Springer-Verlag 1996, 421-4
10. J.L.Fiadeiro and T.Maibaum, "Categorical Semantics of Parallel Program Design", *Science of Computer Programming* 28, 1997, 111-138.
11. J.L.Fiadeiro and A.Lopes, "Semantics of Architectural Connectors", in *TAPSOFT'97*, LNCS 1214, Springer-Verlag 1997, 505-519.
12. J.L.Fiadeiro and A.Lopes, "Algebraic Semantics of Coordination, or what is in a signature?", in *AMAST'98*, A.Haeberer (ed), Springer-Verlag 1999.
13. P.Finger, "Componend-Based Frameworks for E-Commerce", *Communications of the ACM* 43(10), 2000, 61-66.
14. N.Francez and I.Forman, *Interacting Processes*, Addison-Wesley 1996.
15. D.Gelernter and N.Carriero, "Coordination Languages and their Significance", *Communications ACM* 35, 2, pp. 97-107, 1992.
16. J.Gouveia, G.Koutsoukos, L.Andrade and J.Fiadeiro, "Tool Support for Coordination-Based Software Evolution", in *Technology of Object-Oriented Languages and Systems – TOOLS 38*, W.Pree (ed), IEEE Computer Society Press 2001, 184-196.
17. S.Katz, "A Superimposition Control Construct for Distributed Systems", *ACM TOPLAS* 15(2), 1993, 337-356.
18. G.Koutsoukos, J.Gouveia, L.Andrade and J.L.Fiadeiro, "Managing evolution in Telecommunications Systems", submitted, accessible at www.fiadeiro.org/jose/papers.
19. G.Koutsoukos, T.Kotridis, L.Andrade, J.L.Fiadeiro, J.Gouveia and M.Wermelinger, "Coordination Technologies for Business Strategy Support: a case study in Stock Trading", submitted, accessible at www.fiadeiro.org/jose/papers
20. J.Kramer, "Exoskeletal Software", in *Proc. 16th ICSE*, 1994, 366.
21. B.Meyer, "Applying Design by Contract", *IEEE Computer*, Oct.1992, 40-51.
22. M.Wermelinger and J.L.Fiadeiro, "Algebraic Software Architecture Reconfiguration", in *Software Engineering–ESEC/FSE'99*, LNCS 1687, Springer-Verlag 1999, 393-409
23. P.Zave, "Feature interactions and formal specifications in telecommunications", *IEEE Computer* XXVI(8), 1993, 20-30.

Feature Based Composition of an Embedded Operating System Family

Danilo Beuche
University of Magdeburg, Magdeburg
danilo@ivs.cs.uni-magdeburg.de *

Abstract

In this paper we describe our experiences made during the application of feature-model based configuration to an operating system family for deeply embedded targets. Although applying feature modeling proved to be very useful, many problems remain. One major problem, the efficiency of the resulting software system, which is cucrial for deeply embedded software, is discussed.

1 Introduction

Embedded systems are one of the most demanding application domains for software producers. The bandwidth of applications is very wide. One thing is common for almost all applications: the available platform is relatively short on resources. Due to , e.g., cost, energy consumption and heat dissipation issues, the software has to use the resources in an efficient way. Until today, most software for embedded systems is created using assembly language or C. Modern software engineering approaches failed to convince the mass of embedded software designers that there are better ways of developing software for such systems. But due to shorter product development cycles and the increasing competition, ways to produce (better) reusable software are in great demand.

Component based software development promises easier software reuse. But existing component models like CORBA or DCOM require too much resources and cannot be used for small embedded targets. Component approaches suitable for embedded systems must provide the ability to adapt perfectly to the surrounding environment, which means to provide only the required services and use only the smallest possible amount of resources for that.

The PURE operating system family [2] tries to close this gap. PURE is a realization of the program family concept [6] with an object-oriented approach. The ancestor of PURE, the PEACE operating system family [7] for massively parallel computers, dates back to the late 80s. PURE

*This work has been partly supported by the Deutsche Forschungsgemeinschaft (DFG), grant no. SCHR 603/1-1 and the Bundesministerium für Wirtschaft (BMWi), grant no. 01 MS 801/7

is targeted for deeply embedded systems which may only provide small amounts of code and data storage and have usually limited computation power compared to desktop computers. Many deeply embedded applications run on 8 or 16 bit processors. Careful resource usage is therefor one of the biggest problems in this domain. PURE is implemented in C++ and uses many but not all of its features. E.g. exceptions are too expensive (especially in memory usage) in most cases, so they are not used in PURE. Other features like virtual methods are only used when appropriate.

The next section provides a short introduction to the development of the feature model for the operating system family PURE. The section 3 of this paper is dedicated to discuss one of the problems that were encountered when using feature models for the configuration of component based systems, which affects the efficiency of the resulting software system. A possible solution is also presented. Section 4 concludes the paper.

2 The PURE way of Feature Modeling

Initially PURE consisted of about 100 classes, which were configured using boolean C++ preprocessor flags (around 30 at this time). This was not only inconvenient but also very error-prone as not all possible flag settings lead to usable systems. To overcome this situation a good way to express dependencies between flags was sought after. Feature modeling as described in [3] and [5] seemed very promising. The idea of feature domain analysis, to represent the commonalities and differences between the applications in a whole domain instead of concentrating on a single application, is naturally implemented by a program family (or product line [1]).

A feature model for deeply embedded operating system configuration was developed, currently consisting of about 230 features. While the model was developed independently of the structure of the implementation, we recognized that the dependency hierarchy of features as expressed in the model often resembled the class hierarchy used to implement these features.

```
Component("UART16450") {
  Description("Supports UART controller 16450, 16550A, 16750.")
  Parts {
    class("UART16x50") {
      Sources {
        file("src/sys/device/serial", "UART16x50.cc",impl)
        file("include/device/serial", "UART16x50.h",def)
      }
    }
    classalias("UARTPortal") {
      Sources {
        classaliasfile("include/device/serial","UARTPortal.h","UARTPortal")
      }
      Value("IOPortal",Prolog("has_feature('x86',_NT)"))
      Value("MemoryPortal",Prolog("true,!"))
    }
  }
  Restrictions {
    Prolog("has_feature('RS232-Serial',_NT),has_feature('x86',_NT),!")
  }
}
```

Figure 1: Sample component description

The second step was to develop a way to map the feature selections out of the feature model to a member of the operating system family. Each functional extension was modeled as a compo-

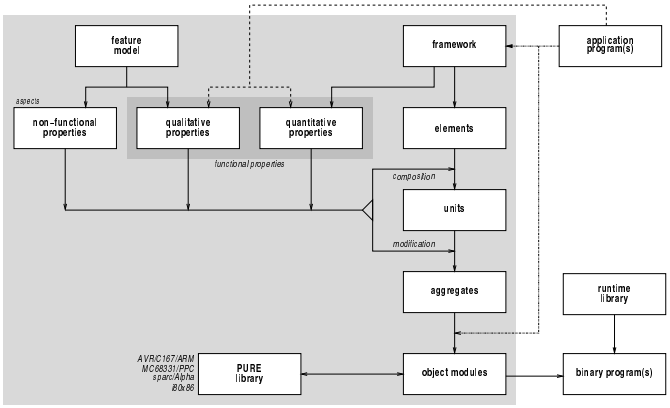


Figure 2: Process diagram for generation of PURE family members

nent. The implementation of a component is not fixed, but generated depending on the selected set of features. There is a set of operations which can be used to realize an implementation. These operations include selection of component parts, implementation files for parts, setting of variables, preprocessor flags or type definitions to calculated values. Each component decides whether it is suitable for a given feature selection or not. A sample component description is shown in figure 1. The configuration rules are expressed and evaluated using Prolog.

The configuration and generation process for a family member is shown in figure 2. First the required set of features has to be selected. Currently this is done manually, but it is also possible to derive certain feature information from an analysis of the target application(s). The resulting family member is then subject to an optimization process which tries to generate the most efficient implementation for a given application or set of applications.

Using the feature model it is now possible to generate family members quite easily and to handle the configuration complexity of such a software system in a better way.

3 Deriving efficient component implementations from feature sets

Although feature models are able to represent the requirements of an application domain, there are problems when it comes to generate efficient applications out of a feature selection from a feature model alone.

3.1 The rs232-serial domain

To illustrate the problem, the small application domain rs232-serial is used. The implementation consists of a single component `rs232driver` only. The component realizes device driver support for 3 different serial controller chips (UART, SCI, SCC). The implementation consists

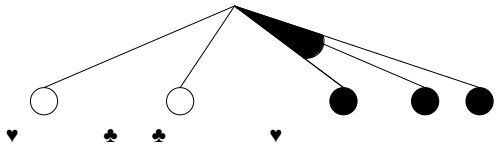


Figure 3: Feature model for domain rs232-serial

of 3 sets of classes realizing the 3 different chip drivers and an optional abstract class which allows to provide a common interface for all drivers. For efficiency reasons the abstract interface class (which introduces virtual methods) should be used only when necessary. Each chip driver has different implementations where the first one is only able to support fixed communication parameter which have to be set at component implementation generation time. The second is able to change the communication parameters at runtime. The third and fourth version extend the first and second implementation by the ability to support interrupt based driver operations. Each implementation has different memory footprint and also processing time requirements. In total, there are at least 68 = $4^3 + 4$ component implementations (3 drivers with each having 4 different implementations and 4 different sets for the common base (UART+SCI, UART+SCC, SCC+SCI, UART+SCC+SCI)) of drivers with a common interface.

The feature model (shown in figure 3) is quite simple and allows to specify the required drivers (at least one has to be selected) and if changeable parameters (changePar) and/or interrupt based operation (interruptOps) are required. It is possible to generate the optimal component implementation purely out of the given feature model only for a subset of the potential usage scenarios. The shown feature model allows only 28 different feature selections. If only one or all of the three chips are required to support the same feature selection out of changePar and interruptOps an optimal implementation can be generated. For most other cases this is not possible.

If two components A and B use the serial driver component, where component A uses the UART driver in interrupt mode and component B uses the SCC driver with changeable parameters, it is not possible to derive an optimal implementation from a feature selection alone. The model allows only to specify that the component has to provide changeable parameters, interrupt operations and must support the UART and SCC chips. Although it is possible to derive an implementation which supports all required functionality as shown on the left side of figure 4, this configuration is not as runtime and space efficient as possible. The optimal implementation is shown on the right side of figure 4.

3.2 How to get the most efficient implementation

A first idea is to extend the feature model so that it becomes possible to make an unambiguous mapping of feature sets to available implementations. But this would lead to a huge number of features in the model, and, more important, the model has to change if implementation changes. This would contradict the idea of representing the application domain with the feature model.

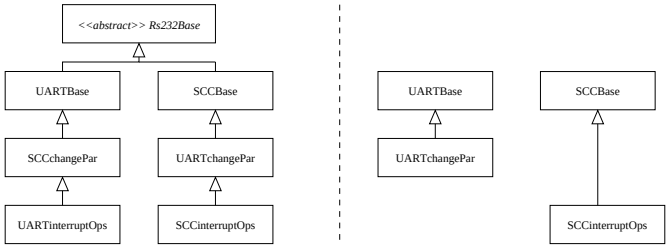


Figure 4: Possible implementations for feature selection changePar,interruptOps,UART,SCC

A better solution is to use separate feature sets for each component which uses the component `rs232driver` and provide a mapping of a set of feature sets to the implementation. The mapping function then becomes a part of the component. This allows the generation of the component implementation independent from the rest of the system.

In the example component A would specify the set UART, interruptOps and component B SCC, changePar. The mapping function should generate for these sets the right hand component implementation as shown in figure 4.

Another solution produces similar results but uses only a single feature set. The idea is to postpone the specialization of the components implementation until system construction time and to use the general purpose implementation initially. A control flow analysis of the complete system shows which component acutally uses what parts of another components interface.

In the example the analysis would reveal that component A creates an object to access the UART driver and enables the interrupt operations, but never changes the communication parameters. For component B similar informations can be gathered from the analysis. This allows to generate an implementation which better suits the usage pattern. This approach requires knowledge about the whole system and the ablility to change the components implementation, while preserving the functionality a using component expects.

While the second approach may fail to generate a working component, the third approach always succeeds. The second approach allows to define more than one feature set, which may or may not be compatible. As the third solution only allows one feature set per component, this can not happen.

For hard real-time deeply embedded application domains the third approach is applicable relatively easy. The systems sources are usually available anyway to allow runtime analysis and for validation and verification purposes.

Using the PUMA C++ analyser and manipulator the restructuring tool BOB [4] based on structure patterns has been built. The first tests show that such an automatic approach is able to generate very efficient implementations out of general component implementations.

In general a combination of the second and third approach is the way to proceed. This allows generation of component implementations without knowledge about the whole system. But for parts of the system for which enough information is available the restructuring can yield even more efficiency.

4 Conclusions

The problem discussed in the paper is one of the most important issues to be solved to make component based software feasible for deeply embedded targets.

However there are many more problems to solve regarding feature based software configuration and composition. One of those problems is the issue how to decide whether a combination of features will lead to a valid system configuration. PURE uses no formal approach for proving validity but relies on dependency rules which have to forbid all invalid cases.

Other problem concern the reuse of existing components and its feature models in other application domains where a different feature model for the application domain is used.

References

[1] Don Batory. Product-line architectures, 1998. Invited presentation, Smalltalk und Java in Industrie and Ausbildung, Erfurt, Germany.

[2] Danilo Beuche, Abdelaziz Guerrouat, Holger Papajewski, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems. In *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'99)*, St Malo, France, May 1999.

[3] K. Czarnecki and U. Eisenecker. *Generative Programming*. Addison-Wesley, Reading, MA, 2000.

[4] Mario Friedrich, Holger Papajewski, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. Efficient Object-Oriented Software with Design Patterns. In *Proceedings of the First International Symposium on Generative and Component-Based Software Engineering (GCSE'99)*, Erfurt, Germany, September 1999.

[5] K. Kang, S. Cohen, J.Hess, W. Peterson, and S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November 1990.

[6] D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–138, 1979.

[7] Wolfgang Schröder-Preikschat. *The Logical Design of Parallel Operating Systems*. Prentice Hall International, 1994. ISBN 0-13-183369-3.

Feature interaction and composition problems in software product lines

Silva Robak
Technical University Zielona Góra, Institute of Computer Science and Management
robak@pz.zgora.pl

Bogdan Franczyk
Intershop Software Entwicklungs GmbH, Intershop Research
b.franczyk@intershop.de

Abstract
Features are essential characteristic of applications within a product line. Features organized in different kinds of diagrams containing hierarchies of feature trees are closely related to variation points, which appear at different levels and life cycle phases for product lines. Optional and alternative variants attached to variation points may by additional constrained by mutual-exclusion or –inclusion and classified by binding modes, times and sites. Variability leads further to crosscutting variability. Reducing an n:m-relation between feature and variation point would help organizing scalable and traceable software models according to SoC with less intertwined feature trees.

Keywords: feature composition, feature interaction, crosscutting features, product lines

Workshop Goals: identify and categorize characterizing properties of various instances of feature interaction problems
Working Groups: feature-based methods for product lines, integrating UML with feature-based methods, handling variability at different levels

Background

Building the sets of related systems helps to achieve the remarkable gains in productivity and to improve time-to-market and product quality [Clements 99]. A development of a group of systems made from common generic software assets is to obtain by building a software product lines upon software product families [Czarnecki 00]. Features are essential characteristic of applications within a product line. Domain Analysis methods [Arrango 94] like feature analysis are on a high level of abstraction and provide a concise and explicit representation of commonality and variability [Coplien 99] contained in product family.

Features may be organized in different kinds of feature diagrams (essential for reuser), containing hierarchies of feature trees (graphs) with mandatory, optional and alternative features [Kang 90]. A root of the tree represents a concept being described and the remaining nodes denote features. Mandatory features have to be always included in every system instance, an optional feature may be included or not, and an alternative feature replaces another feature when included. Mandatory features which parent-features are neither optional nor connected to alternatives represent the common features that each family member possesses. The rest of the features are known as variant features, which represent the permissible differences (deltas) between family members. A part of the implementation process of a member is a selection of these variants. The means like configuration tables

containing the variable features of the member additionally describe the choices, which can be made.

Optional and alternative features are closely related to variation points [Jacobson 97] and may appear at different levels and phases in software development process Domain Engineering (DE) or Application Engineering (AE) for product lines. The later the variability points (seen as “delayed decisions”) would be introduced to a system, the more different systems could be build [Gurp2000].

Variants attached to variation points lead to a set of combinations, but only a subset of it may be correct and leads to complete configurations. The problems associated with feature specification and interaction (i.e. when one feature modifies or influences another features) have been discussed since early nineties [Zave 93]. Variability may be resolved with different techniques, especially efficient with those according to Separation of Concerns (SoC) principle.

Position

Our position is that:

1. There is no concise feature definition, because features of various granularities are requirements (functional and non-functional) and properties of product lines as software intensive systems. The features may be described in different models and views and depend on stakeholders attached to the view (e.g. customer, system analysts and designer, user, end user). The products in a product line are sharing a common, managed set of features that satisfy specific needs of selected market or mission, while product families share features of more technical nature.

2. Essential features may be organized in different kinds of diagrams containing hierarchies of feature trees. Commonly used kinds of feature trees [Kang 90, Kang 98, Griss 98, Czarnecki 00, Gurp 00] are alone not able to express various possible feature’s roles dependencies and relationships. Complementary notations for enhancing semantic of the domain models (like described in [Hein 00]) are needed.

3. Decisions about what will be common and what the variable parts in a software product line have more strategic then technical nature and can therefore change over time. Commonality expressed as mandatory features constraint the size of the software family. Variability as optional or alternative features enlarges the family size, but as generic (i.e. parameterized) places also increase systems complexity. Good design (e.g. according to SoC) helps to avoid crosscutting variability when product line evolves.

4. Optional and alternative feature are closely related to variation points, which may appear at different levels and life cycle phases for product lines. The variability has to be resolved within a particular part of the DE or AE and by decision-makers responsible for it.

5. In feature diagrams optional and alternative variants attached to variation points may by additional constrained by or mutual exclusion (strong form of conflicting [Bosch 00]) or mutual inclusion. Dependencies and composition rule for features may be partly described within feature trees with UML-like relationships (i.e. composed_of,

generalization/ specialization, implemented_by) [Kang98], [Bosch 00], and other means like tables and matrixes, textual descriptions as constraints.

6. Features may by further classified by binding modes, times and sites. Binding time may be classified as construction- time, installation- and use-time. Construction- time (also referred to as a build time) includes the source time (pre-compile time), compile, link and load time. For generative techniques a generation time may be seen as an additional kind of binding time within a build time.

7. Variability at certain design stage leads further to crosscutting variability in later stages. Reducing an n:m-relation between feature and variation point would help organizing scalable and traceable software models according to SoC with less intertwined feature trees. The usage of the generative techniques like Aspect-Oriented Programming AOP [Kiczales 97], Subject-Oriented Programming SOP [SOP] or frame engineering [Basset 97] may play the special role there.

8. Object-oriented technology notations like UML diagrams should be used together with feature models for modeling variants during analysis.

Approach

We can treat features as abstractions of the requirements, too. A particular requirement may apply to several features and a particular feature can be required to fulfill more than one requirement (a n:n-relation) [Gurp2000]. There is usually a 1:n-relation between a feature and its implementation. A feature implementation may be usually spread through many assets ("crosscutting features"), except using special techniques like AOP or SOP "separating concerns".
Domain analysis methods are used to identify and grouping sets of features. The FORM approach [Kang 98] is an extension to FODA [Kang 90] and provides methodology for developing domain architectures and components for reuse upon feature models. Like FeatuRSEB [Griss 98] extended the UML-based RSEB method [Jacobson 97] with the feature model, the integration of feature oriented methods with the object-oriented technology seems to be the next step towards supporting product line development.

The shortcoming of the feature trees is the restriction of variability in feature specification to some binding times and a decomposition type. There can be also other attributes needed to choose a variant like availability sites (i.e. when, where and to whom a feature can be available), variability mechanisms, and binding modes (e.g. static, dynamic), binding occurrence, descriptions, etc.

There are many different approaches for solving variability at the code level. The chosen technique should possible support the SoC-principle and provide the maintainability (traceability) and the scalability of applications. Features should be forwards and backward traceable with different tools helping untangling feature maps from features to products [Griss 00].

Handling variability at the code level

There are several approaches to handling product-line variability at a various levels of abstraction, according to different binding times and associated with them a generic assets

representation. The common object-oriented techniques are abstractions, the different kinds of inheritance, the overloading, and the aggregation with the delegation and also the parameterization. Some of the techniques were derived from a particular programming language and are not available everywhere (i.e. overloading and multiple inheritance is not available in Smalltalk or in Java). Another new techniques like AOP or Dynamic Class Libraries are not obtainable for older programming language (i.e. for C++). The object-oriented techniques like aggregation with delegation are used in design patterns. Application frameworks using design patterns may be employed in product line context since they provide solutions for managing the variations. Mechanisms like static libraries, conditional compilation and generative techniques like frame technology [Basset 97] are possible to use for all programming languages.
The crosscutting variability that affects many components may be difficult to handle.

References

[Arrango 94] Arrango G., *Domain Analysis Methods*. In Software Reusability, Schäfer, Prieto-Díaz R., and Matsumoto M. (Eds.), Ellis Horwood, New York, New York, 1994, pp. 17-49.

[Basset 97] Basset P., *Framing Software Reuse - Lessons from Real World*, Yourdon Press, Prentice Hall, 1997.

[Bosch 00] Bosch J., *Design and Use of Software Architectures. Adopting and evolving product-line approach*. Addison-Wesley, 2000.

[Clements 99] Clements P., Northrop L.M., *A Framework for Software Product Line Practice* - Version 2.0 [online]. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, July 1999. <URL: http://www.sei.cmu.edu/plp/framework.html>

[Coplien 99] Coplien J., *Multi-Paradigm Design for C++*, Addison-Wesley, 1999.

[Czarnecki 00] Czarnecki K., Eisenecker U., *Generative Programming Methods, Tools and Applications*, Addison-Wesley, 2000.

[Griss 98] Griss M. L., Favaro J., D'Alessandro M., *Integrating Feature Modelling with the RSEB*. Proceedings of ICSR98, Victoria, BC, IEEE, June 1998, pp. 36-44.

[Griss 00] Griss M. L., *Implementing Product-Line Features with Component Reuse*. 6th International Conference, proceedings/ICSR-6, Vienna, Austria, June 27-29, 2000; In Wiliam B. Frakes (ed.) "Software Reuse: Advances in Software Reusability". Springer, pp. 137-152.

[Gurp 00] Gurp J., Bosch J., Svahnberg M., *Managing Variability in Software Product Lines*. Landelijk Architectuur Congres 2000.

[Hein 00] Hein A., Schlick M., Vinga-Martins R., *Applying Feature Models in Industrial Settings*. Preceedings of the The First Software Product Line Conference (SPLC1). Denver, Colorado, USA, pp.47-70, 2000.

[Jacobson 97] Jacobson I., Griss M. L. and Jonsson P., *Software Reuse Architecture, Process and Organization for Business Success*, Addison-Wesley, 1997.

[Kang 90] Kang K., Cohen S., Hess J., Nowak W., and Peterson S., *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report No. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 1990.

[Kang 98] Kim S.J., Lee J., Kim K.J., Shin S.H., and Huh M.H., *FORM*: *A Feature-Oriented reuse Method with Domain specific reference architectures.* Annals of Software Engineering, Vol. 5, pp.143-168, 1998.

[Kiczales 97] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J. M., Irvin, J., *Aspect-Oriented Programming*, Proceedings ECOOP97 – 11th European Conference of Object-Oriented Programming, Jyväskylä, Finland, June 1997, Mehmet Aksit and Satoshi Matsuoka (Eds), LNCS 1241, Springer-Verlag, 1997.

[SOP] Homepage of the Subject-oriented Research Project, IBM, Thomas J. Watson Research Center, Yorktown Heights, NY, <URL: http://www.research.ibm.com/sop>

[Zave 93] Zave P., *Feature interactions and formal specifications in telecommunications*, IEEE Computer XXVI (8): pp.20-30, August, 1993.

ECOOP 2001
Workshop *Feature Interaction in Composed Systems*
June 18, 2001 – Budapest, Hungary

Configuring Software Product Line Features

Andreas Hein[1], **John MacGregor**[2], and **Steffen Thiel**[3]

Abstract. The central goal of the Product Line Approach [Cle99] is the systematic reuse of core assets for building related products. At our corporate R&D department for software technology we are concerned with developing, adapting and validating processes, methods, and tools that support the Product Line Approach for Bosch business units. In this paper we present our experience and thoughts on configuring product line features for embedded automotive systems.

1 INTRODUCTION

The essential goal of the Product Line Approach [Cle99] is the systematic reuse of core assets for building related products. Therefore, the Product Line Approach integrates two basic processes: the abstraction of the commonalities and variabilities of the products considered (development *for* reuse), and the derivation of product variants from these abstractions (development *with* reuse). Both processes should be supported by tools because of their complexity in industrial settings [CE00] [LKK+00] [SH00] [TLS+98]. Until now, the product line community has mainly focused on the first process (see, e. g., the FODA approach [KCH+90]). The second process has certainly been addressed by more recent product line engineering methods (e. g. by FORM [KKLK98], an extension of FODA), but is not, as yet, complete to the last detail. In the context of embedded automotive systems, hundreds or even thousands of variants may be produced per year. So many variants make every effort in automating the product derivation process worthwhile, especially when software development capacity is scarce. In particular, support for configuring consistent and practical feature combinations is inevitable for avoiding product specifications that should not or cannot be realized. Actually, the configuration of variants has been a major AI (Artificial Intelligence) topic [GK99]. To support configuration, AI proposes to formalize the domain knowledge so that the derivation of individual products can be partly automated. We think that integrating AI solutions into the product line development *with* reuse process is a promising approach.
For a better understanding of what is required from the development *with* reuse, we will first have a short look at the development *for* reuse. The following is based on a case study in the Car Periphery Supervision (CPS) domain [PRAISE] [TFF+01].

[1] Robert Bosch GmbH, Corporate Research and Development – FV/SLD, P.O. Box 94 03 50, D-60461 Frankfurt am Main – email: andreas.hein1@de.bosch.com
[2] Robert Bosch GmbH, Corporate Research and Development – FV/SLD, P.O. Box 94 03 50, D-60461 Frankfurt am Main – email: john.macgregor@de.bosch.com
[3] Robert Bosch GmbH, Corporate Research and Development – FV/SLD, P.O. Box 94 03 50, D-60461 Frankfurt am Main – email: steffen.thiel@de.bosch.com

2 DEVELOPMENT *FOR* REUSE

Development *for* reuse is concerned with engineering reusable core artifacts. FODA feature models support this process as they explicitly represent those portions of a domain that are common to all or most products and are thus predestined for reuse. Furthermore, the variation points that should be encapsulated to get a stable product line design are also represented explicitly.

2.1 Getting an overview of domain capabilities

A domain feature model gives an overview of the capabilities of product line members, which variants exist and how they depend on each other. Therefore, feature modeling organizes capabilities into a conceptual hierarchy with taxonomies and partonomies. Based on the *is-a* and *is-part-of* relationships, common and variable portions of product line members can be explicitly represented. In addition, feature modeling introduces composition rules to constrain the range of valid feature combinations. Composition rules allow the specification of how variable features from different branches of the conceptual hierarchy may be combined to build consistent and practical product specifications.

2.2 Structuring domains for reuse

Commonality and variability modeling is a prerequisite for identifying reusable components. Starting from the feature overview, the product line architecture can be designed to maximize reusability and configurability. The feature structure is then reflected in the corresponding conceptual architecture. It is unlikely that reusability and configurability are the only qualities in an architecture, however. In the CPS context, performance and reliability also play an important role. The final architecture must consider *all* functional and non-functional requirements, including qualities and design constraints.

3 DEVELOPMENT *WITH* REUSE

A feature model is not only valuable for getting an overview of domain capabilities and structuring a domain for reuse. It may also serve as a starting point for building products by assembling their features. It thus plays an important role in the development *with* reuse process.
It practically goes without saying that it is counter-productive to allow the derivation of products from meaningless or even faulty feature combinations. In order to avoid such combinations, the knowledge about what “meaningless” and “faulty” means must be made explicit. This inevitably leads to more extensive and more complex models. Nevertheless, the question is how consistent and practical feature configurations can be achieved

systematically. Therefore additional knowledge is needed about how the existing options can be effectively combined to satisfy specific requirements. This knowledge must also be so formalized and integrated into the domain model as to make tool support for product lines feasible on an industrial scale.
AI knowledge representation formalisms and concepts such as configuration tasks, strategies, and reasoning seem promising in this respect. Our goal is to enhance feature models with these AI techniques to enable product derivation from product line assets. Therefore, the processes outlined in the following sections must be taken into account.

3.1 Capturing User Requirements in Terms of Features

Feature modeling will most likely be used on different abstraction levels of a product line: while customers or end users want to get an overview in order to decide on specific product options, a systems engineer must have a much more detailed model to be able to compose and parameterize components so that they finally realize the user requirements.
In view of that, customers cannot be directly confronted with the system engineering feature model for several reasons. On the one hand, customers would be overloaded with information irrelevant to their context. On the other hand, customers would probably not find features that directly map to their requirements and notions. And finally, the system engineering knowledge will presumably include company secrets so that it should not be accessible for customers anyway.
Therefore, a detailed domain feature model must be abstracted and those parts that are relevant for customers must possibly be reformulated; in terms of the intended functionality, for example. In principle, individual high-level feature models must be developed for and tailored to every target group whether development engineers reusing components or sales engineers supporting customers. These tailored feature models would then be used to capture the customer requirements in terms of features, and can be compared to *configuration tasks* as proposed by [Gün95].
Of course, the customer feature models must be linked to the more detailed system engineering model, that is the different abstraction levels must be mapped. A customer choice then automatically selects the corresponding features in the engineering model and restricts the solution space accordingly.

3.2 Guiding the User through the Configuration Process

As has been indicated above, variable features are selected in an interactive process with the customer or end user. The order in which the decisions are made should be controlled by *strategies* that take a customer's special interests and skills into account as well as the effectiveness of the configuration process. On the one hand, a sophisticated customer would most likely have concrete views about the type of sensor technology needed for a specific CPS product, whereas an uninitiated end user would probably prefer to make his choices based on the "user-visible" system capabilities. On the other hand, particular sequences of configuration steps lead directly to the configuration goal; for example, by dealing with the fundamental decisions first many detail decisions become irrelevant. Following a specific strategy ensures that a customer will only be presented with choices that are relevant and opportune to the particular configuration stage.
Note that strategies are domain knowledge and thus should be included in the domain model. The feature model abstractions and strategies can be combined to make tailored profiles that guide customers or end users individually through the configuration process.

3.3 Reasoning Based on Conceptual Hierarchy and Constraints

Apart from using feature model abstractions and strategies, development *with* reuse also requires that a feature model be formalized so that automated structure-based *reasoning* can be employed. Nevertheless, existing constraints must also be evaluated in general to keep the configuration consistent. That is, if a customer requires feature A that depends on feature B, feature B should automatically be identified as an obligatory choice. Similarly, if one feature excludes another feature, inconsistencies between the selections of these features should be detected automatically.
The assumption about the completeness of the feature model strongly influences the reasoning that can be done [HMSV00].
Assuming that all product variants have been foreseen makes enhancements of the model superfluous. In other words, the feature model is assumed to be substantially static. This case is also referred to as *routine configuration*. An inconsistency in solving a routine configuration problem can always be resolved by taking back one or more decisions made (*backtracking*) and searching in another direction. Restricting solutions to routine configuration enables strong reasoning support due to the tight limitations of the corresponding search space. Nevertheless, one can imagine that in most contexts there will be requirements that cannot be realized with routine configuration. These cases demand *innovative configuration.*
Innovative configuration is needed for deriving new products which were not considered while building the product line. An innovative problem involves creating new combinations of features which are beyond the scope of the existing model. Backtracking is not inevitable when there is a consistency conflict in an innovative problem, as the feature model can be extended with new features that resolve the problem. The introduction of new features into the domain may cause existing restrictions to be removed.
The type of configuration problem not only affects the reasoning mechanisms, but also is an indicator for the complexity of the software engineering task. This way it is possible to better estimate the effort needed to build a product already in the requirements definition stage.

4 RELATED WORK

Our work is concerned with configurable industrial product lines and is based on FODA [KCH+90] which provides a fundamental means to plan *for* reuse. FORM [KKLK98] extends FODA in that it integrates the development *with* reuse, but lacks an adequate representation and formalization to allow for large-scale production of variants. Therefore, configuration tools that implement AI concepts [Gün95] [GK99] seem to be more promising. Our goal is to combine the Product Line Approach methodology and AI configuration to fit our needs.
Nevertheless, there are other approaches to this issue. The Helsinki University of Technology has developed its own concept for modeling configurable (hardware) product families [TLS+98]. Generative Programming [CE99] [CE00] covers methods and tools for designing and implementing system

families as well as for automating component assembly. The assembly is especially supported by Domain-Specific Languages (DSLs) that allow to define system family members in a specialized, problem-oriented language. Aspect-Oriented Programming (AOP) [AOP] contributes to composing features that crosscut functional components, such as error handling or resource management.

5 CONCLUSIONS

The goal of the Product Line Approach is not only the planned development of reusable components for related products. In the end, the main interest is to build new systems from their requirements by systematically assembling the reusable components. A central issue is the question of how to formalize the product derivation process so that meaningless or faulty combinations of product line features are avoided. The Bosch approach therefore combines the product line and configuration methodologies to enable adequate tool support for the development *with* reuse based on formal domain models.

6 OUTLOOK

This paper has focused on configuring consistent systems from analysis level product line assets. In order to derive implemented products, their requirements must finally be traced to code. This cannot be done in a single step, as there will not be a direct mapping from the problem specification to the implementation structure. Rather, the feature configuration is propagated through a number of conceptual layers that transform the requirements step by step to their final realization.

The product derivation process involves some important issues:

- The feature model representations and product line architectures should be coordinated so that features can be mapped to conceptual architectural components.
- Architectural mechanisms are required to ensure flexible component compositions.
- The decisions made during configuration on the analysis level are not isolated. Therefore, other models (for example, the product line architecture) must be included in the reasoning chain and formalized accordingly.

We are currently investigating the propagation of feature configurations to compose product implementations from abstract product line descriptions.

REFERENCES

[AOP] Homepage of the Aspect-Oriented Programming Project, Xerox Palo Alto Research Center (Xerox PARC), Palo Alto, CA, www.parc.xerox.commonality/aop/

[Cle99] Clements, P.: Software Product Line – A New Paradigm for the New Century; Cross Talk, pp. 20-23, February 1999.

[CE99] Czarnecki, K., and Eisenecker, Ulrich W.: Synthesizing objects. In: Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP'99), Springer, 1999.

[CE00] K. Czarnecki, and Ulrich W. Eisenecker: Generative Programming. Methods, Tools, and Applications. Addison-Wesley, 2000.

[Gün95] A. Günter: KONWERK – ein modulares Konfigurierungswerkzeug. Universität Hamburg, FB Informatik. 1995.

[GK99] A. Günter, and C. Kühn: Knowledge-Based Configuration – Survey and Future Directions. XPS-99, Knowledge-Based Systems, Würzburg, Germany.

[HMSV00] A. Hein, J. MacGregor, M. Schlick, and R. Vinga-Martins: PRAISE: Lessons Learned. CEC Deliverable P28651-D3.4, ESPRIT Project PRAISE, March 2000.

[KCH+90] Kang, Kyo C., Cohen, Sholom G., Hess, James A., Novak, William E., and Peterson, A. Spencer: Feature-Oriented Domain Analysis (FODA). Feasibility Study. Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University, Software Engineering Institute, 1990.

[KKLK98] K. C. Kang, S. Kim, J. Lee, and K. Kim: FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. Annals of Software Engineering, Vol. 5, pp. 143-168, 1998.

[LKK+00] K. Lee, K. C. Kang, E. Koh, W. Chae, B. Kim, and B. W. Choi: Domain Oriented Engineering of Elevator Control Software. A Product Line Practice. In: P. Donohoe (ed.): Software Product Lines – Experience and Research Directions. Proceedings of the First Software Product Line Conference (SPLC1), August 28-31, 2000, Denver, Colorado, USA. Kluwer Academic Publishers, pp. 3-22, 2000.

[PRAISE] ESPRIT Project 28651: PRAISE – Product-line Realisation and Assessment in Industrial Settings. IT RTD Project Programme, 1998, http://www.esi.es/Projects/Reuse/Praise/

[SH00] M. Schlick, and A. Hein: Knowledge Engineering in Software Product Lines. European Conference on Artificial Intelligence (ECAI 2000), Workshop on Knowledge-Based Systems for Model-Based Engineering, August 22, 2000, Berlin, Germany.

[TFF+01] S. Thiel, S. Ferber, T. Fischer, A. Hein, and M. Schlick: A Case Study in Applying a Product Line Approach for Car Periphery Supervision Systems. In: Proceedings of In-Vehicle Software 2001, pp. 43-55, SAE International Congress 2001. March, 5-8, 2001, Detroit, Michigan, USA.

[TLS+98] J. Tiihonen, T. Lehtonen, T. Soininen, A. Pulkkinen, R. Sulonen, and A. Riitahuhta: Modeling Configurable Product Families. 4th WDK Workshop on Product Structuring, October 22-23, 1998, Delft University of Technology, The Netherlands.

Representing and Reasoning on Feature Architecture: A Description Logic Approach

Yu Jia[1] Yuqing Gu[2]
Institute of Software, Chinese Academy of Science
Beijing 100080, China
Email: [1]jia_yu@263.net [2]yuqing_gu@isss.iscas.ac.cn

Abstract: *In Component-based Development (CBD), the feature-oriented method is regarded as a good means to represent Domain-Specific Software Architecture (DSSA). However, the graphical and textual description of feature architecture is not rigorous and lacks of reasoning mechanism for feature relationships. In this paper we address these problems by abstracting the relationship types between features and by formalizing these types in the Description Logics (DLs). As an application, we present a DLs-based reasoning procedure for the feature interaction problem.*

1. Introduction

The Feature-Oriented Reuse Method (FORM) [Kang98] is the way of describing software functional and non-functional properties by domain features, which are the well-defined truth understood by both users in problem domain and developers in solution domain. The "buying and integrating" philosophy of Component-Based Development (CBD) [Alan00] requests a directly mapping between problems and solutions when retrieving and matching components [Penix95], and also the compositable component semantics supporting adaptation and integration [Blom00]. The internal relationship between FORM and CBD makes "features" to be reconsidered and used as the constituent elements to describe the component semantics. This not only overcomes the difficulty of component semantic representation in CBD, but also brings a benefit that the feature-oriented Domain Engineering (DE) provides researchers and practitioners an opportunity to describe the Domain-specific Software Architecture (DSSA) in perspective of Knowledge Representation (KR). That is, using the features-based knowledge describing languages as tools to indicate the commonality and variability among product families.

Although FORM is regarded as the only practical method in DE, to date its feature model is described in a kind of extended AND/OR graph or the textual specification language [Kang98] which is ambiguous and inaccurate, let alone there exists the automatic inference on features relationships which is radically needed in operations as checking feature interaction, subsumption etc.

In this paper, Section 2 presents a set of essential feature architecture concepts for overall feature-oriented methodology. Then in section 3 some general relationship types are abstracted, which are independent of particular fields, and are described in an expressive Description Logic language [Calvanese99,Donin97] called $\mathcal{FDL}$ in section 4. Finally in section 5 we propose a DLs-based reasoning approach for the

feature interaction problem through the feature relationships in the knowledge base.

2. The Conceptual Model of Feature Architecture

The concept of "feature" is not new in computer science. A variety of definitions have given in very different ways from different fields such as "A distinguishing characteristic of a software item, for example, performance, portability, or functionality " [IEEE 90], or "A feature in the context of telephone systems is an addition of functionality to provide new behavior to the uses or the administration of the telephone system" [Araces 99]. However, it is evident that these definitions are informal and inconsistent, as well as have not revealed the essence of feature. With the rising recognize of software reuse is in fact the knowledge reuse. We believe that the concept of feature can only be captured in the view of Knowledge Representation. That is, the role of feature is to state the domain knowledge and provide a common agreement or contract to a specific domain.

The conceptual model for the feature-oriented methodology may be constituted by four elements: a name convention, an interpretation rule, a meta knowledge base and a relationship constrain.

Name convention – How to name a feature, including the set of allowed symbols and constructing regulations.

Interpretation rule – Defining the mapping mode between feature name and domain knowledge, such as whether many feature names can match one meta knowledge assertion.

Meta knowledge specification mechanism – Stating the truth in the problem domain for a specific feature. The natural language is used in most occasions.

Relationship constrain – Defining the relation types among features to form the composition rules. Relationships should be clearly and unambiguously defined in formal methods.

Based on above conceptual model, we present the formal definition of Feature and Feature Architecture.

DIFINITION 2.1 (*Feature*) A feature $f = (n, k)$ where n is the name of feature which follows the name convention and k is the interpretation of n in language of meta knowledge for a specific problem domain.

EXAMPLE 2.2 In the telephone domain, there exists a feature: n = "Call Waiting" and k = "permitting the subscriber to accept a send call when the telephone is already in use". Here natural language is used to describe the domain knowledge.

DIFINITION 2.3 (*Feature Architecture*) A Feature Architecture $\Omega = (F, R, K, I_f, I_r)$

Where F is a set of features,

R is a set of feature relations in form of $F \times F$,

K is a set of meta knowledge with constitutes a knowledge base,

I_f is a function interpreting the meaning of features: $F \rightarrow K$,

I_r is a function interpreting the meaning of feature relations: $F \rightarrow R$.

To a given domain Ω is the DSSA, while the semantics of software system is considered as an instance of the Ω.

3.The Feature Relationship Types

A high level understanding of the feature relationship is required when selecting the formal method for describing the feature architecture. That is, a formal language should be expressive enough to describe all possible relationship categories independent of specific application domain. Following we define four distinct relationship types which are considered as the underpinning of the feature architecture.

Aggregation relationship – An aggregation (also called *composition*) represents a part-whole relationship. That is, a feature (called parent) composed of two or more features (each called a child). *Requisite aggregation* means parent exists in condition of child exists. *Optional aggregation* means a child can exist or not to its parent.

Generalization relationship – Generalization is a relationship between a general feature (called abstract feature) and a specific feature (called refined feature) that the specific feature is consistent with the general feature in semantics and contains more detail. Generalization is a transitive, antisymmetric relationship. *Refinement* (also called *specialization*) is the opposite relationship to the generalization.

Dependency relationship – A dependency indicates a semantic relationship between two or more features. There are several particular kinds of dependency: *Conditional dependency* says a feature can change the behavior or properties of another feature; *Conflict dependency* means a feature semantically conflicts with another feature such that two features can not be both contained in a feature architecture. Note that in aggregation relationship the parent has conditional dependency relation with its child.

Association relationship – An association describes two or more features share a same semantic part; that is, associated features are intersected. *Equivalence* is special association relationship indicates that features have the same semantics. *Isolation* is the opposite relationship to the association which means two features have no relationship.

4. Representing Feature Architecture in An Expressive DL

Description Logics (DLs) is a well-defined Terminological Knowledge Representation Language with the set-theoretic semantics. In DLs the domain of interest is modeled by means of *individuals*, *concepts*, *roles* and *knowledge base* exactly corresponding to the *meta knowledge specifications*, *features*, *feature relationships* and *feature architecture* respectively. In the rest of this paper, we use these corresponding terms without any distinction.

In table 4.1 we show a kind of DLs called $\mathcal{FDL}$, especially for expressing feature architecture. In DLs, starting from a set of *atomic concepts* and *atomic roles*, one can build complex concepts and roles by applying certain *constructs*. We denote atomic concepts by $\boldsymbol{A}$, arbitrary concepts by $\boldsymbol{C}$ and $\boldsymbol{D}$, atomic roles by $\boldsymbol{P}$, all possibly with subscripts. We also use the following abbreviations to increase readability: $\bot$ for $\neg\top$, $\boldsymbol{C}_1 \sqcup \boldsymbol{C}_2$ for $\neg(\neg\boldsymbol{C}_1 \sqcap \neg\boldsymbol{C}_2)$, and $\exists\boldsymbol{P}.\ \boldsymbol{C}$ for $\neg\forall\boldsymbol{P}.\neg\boldsymbol{C}$ (means $\{\boldsymbol{o} \mid \exists\boldsymbol{o}' : (\boldsymbol{o}, \boldsymbol{o}') \in \boldsymbol{P}^{\mathcal{I}}\}$). Note

that arbitrary role is not allowed in $\mathcal{FDL}$.

In DLs the formal semantics is specified through the notion of interpretation. An *interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ consists of a set $\Delta^{\mathcal{I}}$ (the domain of $\mathcal{I}$) and a function $\cdot^{\mathcal{I}}$ (the interpretation function of $\mathcal{I}$) that maps every concept to a subset of $\Delta^{\mathcal{I}}$(i.e. $\mathbf{C}^{\mathcal{I}}$ to concept $\boldsymbol{C}$); and every role to a subset of $\Delta^{\mathcal{I}}\times\Delta^{\mathcal{I}}$ (i.e. $\boldsymbol{R}^{\mathcal{I}}$ to role $\boldsymbol{R}$), respecting the specific conditions imposed by the structure of the concept or role.

Concepts $\boldsymbol{C}$	Syntax	Semantics
atomic concept	$\boldsymbol{A}$	$\boldsymbol{A}^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
universal concept	$\top$	$\Delta^{\mathcal{I}}$
negation	$\neg\boldsymbol{C}$	$\Delta^{\mathcal{I}} \setminus \boldsymbol{C}^{\mathcal{I}}$
intersection	$\boldsymbol{C}_1 \sqcap \boldsymbol{C}_2$	$\boldsymbol{C}_1^{\mathcal{I}} \cap \boldsymbol{C}_2^{\mathcal{I}}$
universal role quantification	$\forall \boldsymbol{P}.\, \boldsymbol{C}$	$\{\boldsymbol{o} \mid \forall \boldsymbol{o}' : (\boldsymbol{o}, \boldsymbol{o}') \in \boldsymbol{P}^{\mathcal{I}} \rightarrow \boldsymbol{o}' \in \boldsymbol{C}^{\mathcal{I}} \}$
quantified number restriction	$(\leq n\, \boldsymbol{P}.\boldsymbol{C})$	$\{\boldsymbol{o} \mid \#\{\boldsymbol{o}' \mid (\boldsymbol{o}, \boldsymbol{o}') \in \boldsymbol{P}^{\mathcal{I}} \wedge \boldsymbol{o}' \in \boldsymbol{C}^{\mathcal{I}}\} \leq n \}$
Roles $\boldsymbol{P}$	Syntax	Semantics
atomic role	$\boldsymbol{P}$	$\boldsymbol{P}^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
concatenation	$\boldsymbol{P}_1 \circ \boldsymbol{P}_2$	$\boldsymbol{P}_1^{\mathcal{I}} \circ \boldsymbol{P}_2^{\mathcal{I}}$
reflect transitive closure	$\boldsymbol{P}^*$	$(\boldsymbol{P}^{\mathcal{I}})^*$
transitive closure	$\boldsymbol{P}^+$	$(\boldsymbol{P}^{\mathcal{I}})^+$
identity	$id(\boldsymbol{C})$	$\{ (\boldsymbol{o}, \boldsymbol{o}) \mid \boldsymbol{o} \in \boldsymbol{C}^{\mathcal{I}} \}$

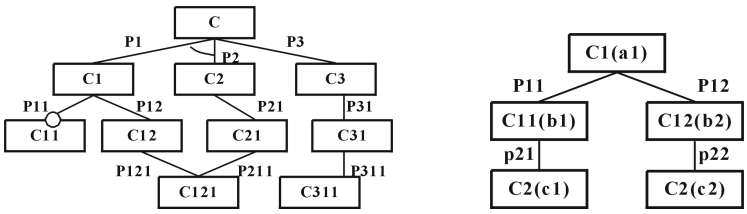
Table 4.1 Syntax and semantics of $\mathcal{FDL}$ concept and role constructs

The feature architecture can be well-formalized on condition that the formal language is powerful enough to describe all possible feature relationship types. Each feature relationship type is expressed in $\mathcal{FDL}$ as shown in Table 4.2, where we denote subsumption by $\sqsubseteq$ ($\boldsymbol{C} \sqsubseteq \boldsymbol{D}$ if $\mathbf{C}^{\mathcal{I}} \subseteq \boldsymbol{D}^{\mathcal{I}}$), and equivalence by $\equiv$ ($\boldsymbol{C} \equiv \boldsymbol{D}$ abbreviates for $\boldsymbol{C} \sqsubseteq \boldsymbol{D}$ and $\boldsymbol{D} \sqsubseteq \boldsymbol{C}$, that is $\mathbf{C}^{\mathcal{I}} = \boldsymbol{D}^{\mathcal{I}}$). Intensional knowledge about features and relations can be expressed through the notion of TBox.

Relationship Types	Expressions	Explanations
Aggregation relationship		
Requisite aggregation	$\boldsymbol{C} \equiv \boldsymbol{C}_1 \sqcup \boldsymbol{C}_2$	$\boldsymbol{C}$ consists of $\boldsymbol{C}_1$ and $\boldsymbol{C}_2$.
Optional aggregation	$\boldsymbol{D} \sqsubseteq \exists \boldsymbol{P}\,.\, \boldsymbol{C} \sqcap (\leq 1\, \boldsymbol{P})$	$\boldsymbol{C}$ subsumes $\boldsymbol{D}$ over $\boldsymbol{P}$ or not .
Generalization relationship	$\boldsymbol{C} \sqsubseteq \boldsymbol{D}$	$\boldsymbol{C}$ is a $\boldsymbol{D}$.
Dependency relationship		
Conditional dependency	$\boldsymbol{C} \sqsubseteq \boldsymbol{P}.\boldsymbol{D}$	$\boldsymbol{C}$ is subsumes by $\boldsymbol{D}$ over $\boldsymbol{P}$.
Conflict dependency	$\boldsymbol{C}_1 \sqsubseteq \boldsymbol{C}$	$\boldsymbol{C}_1$ is conflict with $\boldsymbol{C}_2$.
	$\boldsymbol{C}_2 \sqsubseteq \neg\boldsymbol{C}$	
Association relationship		
Equivalence	$\boldsymbol{C} \equiv \boldsymbol{D}$	$\boldsymbol{C}$ and $\boldsymbol{D}$ are equivalent.
Isolation	$\boldsymbol{C} \equiv \boldsymbol{C}_1 \sqcap \boldsymbol{C}_2$	$\boldsymbol{C}_1$ and $\boldsymbol{C}_2$ are disjoint.
	$\boldsymbol{C} \sqsubseteq \bot$	

Table 4.2 Expressions for Feature Relationship Types

EXAMPLE 4.1 Figure 4.1 show a feature architecture depicted as the feature tree (strictly not a pure tree) which visually presents hierarchy relationships. Here, the little circle denotes optional relation; the arc denotes alternative relation. .



(a) DSSA (b) Component

Figure 4.1 An Example of Feature Architecture

The above feature architecture can be expressed in $\mathcal{FDL}$ as follows:

(a) Σ = { C $\equiv \exists$P1.C1$\sqcap\exists$P3.C3$\sqcup\exists$P2.C2$\sqcap$P3.C3, C1 $\equiv \exists^{\leq 1}$ P11.C11$\sqcap$P12.C12,
C2 $\equiv$ P21.C21, C3 $\equiv$ P31.C31, C12 $\equiv$ P121.C121, C21 $\equiv$ P211.C121,
C31$\equiv$P311.C311 }

(b) Σ = {C1(a1), C11(b1), C12(b2), C2(c1), C2(c2), P11(a1, b1), P12(a1, b2),
p21(b1, c1), p22(b2, c2)}

5. Reasoning Feature Interaction in $\mathcal{FDL}$

The formal method describing the feature architecture in $\mathcal{FDL}$ provides us the possibility to fulfill the task of reasoning feature interaction, which is informally defined as the behavior of one feature influencing the behavior of another. Revealed by the $\mathcal{FDL}$, the feature interaction arises from a *clash* in a *complete* (see below) system generated from two interacted features.

Supporting there exist two arbitrary concepts (i.e. features) $\boldsymbol{C}$ and $\boldsymbol{D}$, an atom concept $\boldsymbol{A}$, and an atom role $\boldsymbol{P}$, we use the calculus introduced in [Donin 97] to design a procedure for detecting interaction between $\boldsymbol{C}$ and $\boldsymbol{D}$.

STEP 1 Creating a conjunction concept $\boldsymbol{E}$ of $\boldsymbol{C}$ and $\boldsymbol{D}$:

$$\boldsymbol{E} \equiv \boldsymbol{C} \sqcap \boldsymbol{D}$$

STEP 2 Transforming $\boldsymbol{E}$ into *negation normal form* $\boldsymbol{F}$ which contains only complements of the atom concept by following ten rules:

(1) $\neg\top\rightarrow\bot$;
(2) $\neg\bot\rightarrow\top$;
(3) $\neg(\boldsymbol{F}_1 \sqcap \boldsymbol{F}_2)\rightarrow\neg\boldsymbol{F}_1 \sqcup \neg\boldsymbol{F}_2$;
(4) $\neg(\boldsymbol{F}_1 \sqcup \boldsymbol{F}_2)\rightarrow\neg\boldsymbol{F}_1 \sqcap \neg\boldsymbol{F}_2$;
(5) $\neg\neg\boldsymbol{F}\rightarrow\boldsymbol{F}$;
(6) $\neg(\forall\boldsymbol{P}.\boldsymbol{F})\rightarrow\exists\boldsymbol{P}.\neg\boldsymbol{F}$;
(7) $\neg(\exists\boldsymbol{P}.\boldsymbol{F})\rightarrow\forall\boldsymbol{P}.\neg\boldsymbol{F}$;
(8) $(\leq n\,\boldsymbol{P})\rightarrow\neg(\geq n+1\,\boldsymbol{P})$;
(9) $\neg\,(\leq n\,\boldsymbol{P})\rightarrow\forall\boldsymbol{P}.\,\bot$ if $n = 1$;
(10) $\neg\,(\leq n\,\boldsymbol{P})\rightarrow(\leq n - 1\,\boldsymbol{P})$ if $n = 1$.

STEP 3 Generating all complete constraint systems deriving from $\{x{:}\boldsymbol{F}\}$.

First we introduce what are the forms of constraint. Assuming x, y, z are variable symbols, $\boldsymbol{\alpha}$ is a function maps every variable to an element of $\Delta^{\mathcal{I}}$, then a constraint is

a syntactic object of one of the forms:

$$x : \boldsymbol{F}, \text{ if } \alpha(x) \in \boldsymbol{F}^{\mathcal{I}};$$
$$x\ \boldsymbol{P}\ y, \text{ if } (\boldsymbol{\alpha}(x), \boldsymbol{\alpha}(y)) \in \boldsymbol{P}^{\mathcal{I}};$$
$$x \neq y, \text{ if } \boldsymbol{\alpha}(x) \neq \boldsymbol{\alpha}(y)$$

A *constraint system* $\boldsymbol{S}$ is a finite, nonempty set of constrains. Following six quasi-completion rules are given to generate constraint systems:

(1) Intersection: $\boldsymbol{S} \rightarrow_{\sqcap} \{x: \boldsymbol{F}_1, x:\boldsymbol{F}_2\} \cup \boldsymbol{S}$, if $x: \boldsymbol{F}_1 \sqcap \boldsymbol{F}_2$ is in $\boldsymbol{S}$, and $x: \boldsymbol{F}_1$ and $x: \boldsymbol{F}_2$ are not both in $\boldsymbol{S}$.

(2) Union: $\boldsymbol{S} \rightarrow_{\sqcup} \{x: \boldsymbol{F}\} \cup \boldsymbol{S}$, if $x: \boldsymbol{F}_1 \sqcup \boldsymbol{F}_2$ is in $\boldsymbol{S}$, neither $x: \boldsymbol{F}_1$ nor $x:\boldsymbol{F}_2$ is in $\boldsymbol{S}$, and $\boldsymbol{F} = \boldsymbol{F}_1$ or $\boldsymbol{F} = \boldsymbol{F}_2$.

(3) Existential Quantification: $\boldsymbol{S} \rightarrow_{\exists} \{x\boldsymbol{P}y, y: \boldsymbol{F}\} \cup \boldsymbol{S}$, if $x:\exists \boldsymbol{P}.\boldsymbol{F}$ is in $\boldsymbol{S}$, there is no z such that z is successor of x and $z: \boldsymbol{F}$ is in $\boldsymbol{S}$, and y is a new variable.

(4) Universal Quantification: $\boldsymbol{S} \rightarrow_{\forall} \{y: \boldsymbol{F}\} \cup \boldsymbol{S}$, if $x:\forall \boldsymbol{P}.\boldsymbol{F}$ is in $\boldsymbol{S}$, y is successor of x in $\boldsymbol{S}$, and $y: \boldsymbol{F}$ is not in $\boldsymbol{S}$.

(5) At-least Restriction: $\boldsymbol{S} \rightarrow_{\geq 1} \{x\boldsymbol{P}y\} \cup \boldsymbol{S}$, if no other completion rule applies to $\boldsymbol{S}$, $x: (\geq n\ \boldsymbol{P})$ is in $\boldsymbol{S}$, x does not have a $\boldsymbol{P}$-successor in $\boldsymbol{S}$, and y is a new variable.

(6) At-most Restriction: $\boldsymbol{S} \rightarrow_{\leq} \boldsymbol{S}[y/z]$, if $x: (\leq n\ \boldsymbol{P})$ is in $\boldsymbol{S}$, x has more than n successor in $\boldsymbol{S}$, and y, z are two $\boldsymbol{P}$-successors of x that are not separated. ($\boldsymbol{S}[y/z]$ denotes the constraint system obtained from $\boldsymbol{S}$ by replacing each occurrence of y by z.)

STEP 4 Checking whether all constrain systems contain a clash. If any of them contains a clash, then $\boldsymbol{C}$ and $\boldsymbol{D}$ are interaction; otherwise $\boldsymbol{C}$ and $\boldsymbol{D}$ are not interaction.

A clash is a constraint system having one of following forms:

(1) $\{x:\bot\}$;

(2) $\{x: \boldsymbol{A}, x:\neg\boldsymbol{A}\}$;

(3) $\{x: (\geq m\ \boldsymbol{P}), x: (\leq n\ \boldsymbol{P})\}$ where $m>n$.

In fact the $\mathcal{FDL}$ is a kind of $\mathcal{ALCN}$. According to [Donin97], above quasi-completion rules provide a soundness and completeness for deciding the satisfiability of $\mathcal{ALCN}$-concepts. The satisfiability of such concepts can be decided in nondeterministic polynomial time. So we conclude that checking the feature interaction has the same computational properties.

6. Conclusion

In this paper we have provided a formal analysis of the feature architecture and the reasoning procedure for feature interaction detection. The similar method is introduced in [Araces99] but we have studied in a more wide and abstract sense. In the future we will integrate our method with standard software component model such as CORBA Component Model (CCM) to expressing and reasoning the semantic relationships among the components.

References

[Alan00] Alan W. Brown: Large-Scale Component-Based Development, Prentice Hall, Inc., (2000)

[Araces99] C. Araces, W. Bouma, and M. de Rijke. Description Logics and Feature Interaction. *Proceedings of the International Workshop on Description Logics - DL-99*: pp 28-32. 1999.

[Blom00] Martin Blom, Eivind J. Nordby. Semantic Integrity in Component Based Development. Project Report, Mälardalen University, Sweden, March 2000.

[Calvanese99] D. Calvanese, G.D. Giacomo, M. Lenzerini, D. Nardi: Reasoning in Expressive Description Logics. In: Alan Robinson, Andrei Voronkov (eds.): Handbook of Automated Reasoning. Elsevier Science Publishers (North- Holland), Amsterdam(1999)

[Donin97] F. M. Donin, M. Lenzerini, D. Nardi, and W. Nutt. The Complexity of Concept Languages. Information and Computation.: pp.1-58, Vol. 34. 1997

[IEEE 90] Standards Coordinating Committee of the IEEE Computer Society. IEEE Standard Glossary of Software Engineering Terminology . IEEE Std 610.12-1990, December 1990.

[Kang98] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, M. Huh: FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. Annals of Software Engineering 5: 143-168. 1998.

[Penix95] John Penix, Phillip Baraona, Perry Alexander. Classication and Retrieval of Reusable Components Using Semantic Features. In Proc: 10th Knowledge-Based Software Engineering Conf., Boston, MA: IEEE Comp. Soc Press, November 1995. 131-138,

[Turner99] C. R. Turner, A. Fuggetta, L. Lavazza and A. L. Wolf. A Conceptual Basis for Feature Engineering. Journal of Systems and Software: Volume 49, Issue 1. 15 December 1999.

Features and Feature Interactions in Software Engineering using Logic

Ragnhild Van Der Straeten
rvdstrae@vub.ac.be
System and Software Engineering Lab
Vrije Universiteit Brussel, Belgium

Johan Brichau*
jbrichau@vub.ac.be
Programming Technology Lab
Vrije Universiteit Brussel, Belgium

Abstract

Feature interactions are common when composing a software unit out of several features. We report on two experimental approaches using logic to describe features and feature interactions. The first approach proposes description logic as a formalization of feature models which allow reasoning about features. In the second approach, a metalevel representation of the software is proposed to capture conditions on features. These conditions are written in terms of the software's implementation providing a uniform formalism that can be applied to any software unit.

1 Introduction

The concepts *feature* and *feature interaction* originated in the telephony-domain. In this context, a feature is an addition of functionality to the basic telephone system providing new behaviour. *Feature interaction* occurs when the behaviour of one feature influences the behaviour of another. When features interact in an unwanted way, a *feature interference* occurs [12].

A *feature* in software engineering can be seen as a concern of the software application [7]. The composition of features will always lead to interactions between features. A *feature interference* occurs when existing or new features interact such that a feature does not behave correctly. This paper discusses two experiments about feature interactions in software engineering.

In the first section we will describe our first approach, in which Description Logic is introduced to formally specify features in the problem domain.

*Research Assistant of the Fund for Scientific Research - Flanders (Belgium) (F.W.O.)

This logic and its reasoning mechanisms are already used to detect feature interactions in the telecommunication domain [1, 3, 4]. In our approach, however, we want to support feature descriptions and interaction detection for the *feature modeling* and *configuration* in domain engineering and in Generative Programming [6]. We want to start a discussion about which information should be described and which reasoning tasks a feature modeling tool should provide.

The next section will introduce our second approach, in which we use Logic Meta-Programming to detect feature interactions in the solution domain. We model features of a system using a logic metalevel representation of the system's implementation. Additional conditions and constraints about the implementation's structure can also be expressed using this metalevel structure. Adding new software artifacts (implementing a new feature) to the system will change the metalevel representation. A feature interference will lead to falsification of the conditions and constraints (imposed by the developer).

2 Description Logics for Feature Modeling

The general idea is to use Description Logic (DL) to formally specify features. As an experimental approach, we initiated the development of a language capturing feature modeling as it is used in Generative Programming [6]. Feature interaction is specified by the dependencies between the different features and the constraints applied on them.

2.1 Description Logics

The family of Description Logics originated from knowledge representation research in Artificial Intelligence. Their main strength comes from the different reasoning mechanisms they offer. The complexity of reasoning in these different languages is and has been widely investigated. These languages have been applied at an industrial level.

The basic elements of a Description Logic are *concepts* and *roles*. A *concept* denotes a set of individuals, a *role* denotes a binary relation between individuals. Arbitrary concepts and roles are formed starting from a set of atomic concepts and atomic roles applying concept and role constructors. For an introduction to Description Logics we refer to [2].

2.2 Feature Modeling using DL

We want to start a discussion about which information should be described and which reasoning tasks a feature modeling tool should provide. The latter is covered in the next section. Feature models appear in the Feature-Oriented Domain Analysis method (FODA) [8] and are used as such by Generative Programming [6]. In this context, a feature model consists of a feature diagram and additional information. This information consists of descriptions of each feature, rationales for each feature, stakeholders and client programs, examples of systems with a given feature, constraints, default dependency rules, availability and binding sites, binding modes, open/closed attributes and priorities. The aspect configuration in [9] can also be interpreted as the modeling (i.e. configuration) of features (i.e. aspects). This section introduces a basic feature language $\mathcal{FML}$ describing the semantics of features and some constraints. $\mathcal{FML}$ is based on the DL $\mathcal{ALCQ}$.

2.2.1 Syntax and Semantics of the Logic

In $\mathcal{ALCQ}$ concepts (denoted by C, D) are formed as follows:

$$C, D \quad \longrightarrow \quad A \mid \neg C \mid C \sqcap D \mid \exists R.C \mid (\leq 1R) \mid \exists^{\leq n} R.C \mid \exists^{\geq n} R.C$$

where A denotes an atomic concept, R denotes an atomic role and n denotes a strict positive integer. The following abbreviations are used: $\top$ for $A \sqcup \neg A$, $C \sqcup D$ for $\neg(\neg C \sqcap \neg D)$, $\forall R.C$ for $\neg\exists R.\neg C$, $\exists^{=n} R.C$ for $\exists^{\leq n} R.C \sqcap \exists^{\geq n} R.C$, $\exists R^{=n|\leq n|\geq n}$ for $\exists R^{=n|\leq n|\geq n}.\top$, $CxorD$ for $(C \sqcup D) \sqcap \neg(C \sqcap D)$. Descriptive semantics, defined by an interpretation function, are adopted, see [10]. A *knowledge base* $\mathcal{K}$ in $\mathcal{ALCQ}$ is a pair $< T, A >$ such that:

- T is the T(erminological)-Box, a finite, possibly empty set of expressions of the form $C_1 \sqsubseteq C_2$ where C_1, C_2 are concepts. This inclusion specifies that C_2 only gives necessary conditions for being an instance of C_1. $C_1 \doteq C_2$ is equivalent to $C_1 \sqsubseteq C_2$ and $C_2 \sqsubseteq C_1$. The formulas in the T-Box are called terminological axioms.

- A is the A(ssertional)-Box, a finite, possibly empty set of expressions of the form $a : C$ or $(a, b) : R$ where C is a concept, R is a role and a, b are individuals.

No restrictions are posed on the terminological axioms. This means that each atomic concept may appear more than once at the left side of an axiom. The terminological axioms may contain cycles, i.e. the concept in the right part of the axiom may refer to the concept in the left part of the axiom.

The *feature language* $\mathcal{FML}$ is completely based on the DL $\mathcal{ALCQ}$. We fix a signature $\Sigma =< Con, Rol, Ind >$, where Con is a countable set of atomic concepts, Rol is a countable set of atomic roles and Ind is a countable set of individuals. A $\mathcal{FML}$ feature model is a set of terminological axioms. The ABox is empty in $\mathcal{FML}$. A feature diagram can be translated to axioms of $\mathcal{FML}$. The set Con consists of all concepts corresponding to the nodes of the diagram. The set Rel consists of all the roles corresponding to the edges of the diagram. The edge decorations are translated using the concept constructors of $\mathcal{ALCQ}$. Consider as an example the following feature description of a car [6]. *A car consists of one transmission and one horsepower and optionally an airconditioning. The transmission is manual or automatic but cannot be both.* This feature model expressed in $\mathcal{FML}$ is shown in figure 1.

$$
\begin{array}{lcl}
\texttt{TRANSMISSION} & \sqsubseteq & \exists^{=1}\texttt{man.MANUAL} \text{ xor } \exists^{=1}\texttt{aut.AUTOMATIC} \\
\texttt{CAR} & \sqsubseteq & \exists^{=1}\texttt{trans.TRANSMISSION} \sqcap \exists^{=1}\texttt{power.HORSEPOWER} \\
 & & \sqcap((\leq 1\texttt{airco}) \sqcap \forall\texttt{airco.AIRCONDITIONING}
\end{array}
$$

Figure 1: The Knowledge Base Corresponding to Features of a Car.

Cardinality constraints. Cardinality constraints can be expressed in the feature model language $\mathcal{FML}$. The constructors $\exists^{\leq n}$ and $\exists^{\geq n}$ admit these kinds of constraints.

If-then constraints. If-then constraints can be integrated in the concept definitions[1]. The constraint *"if there is airconditioning in a car then the horsepower of the car must be greater than or equal 100"*, can be written down as follows[2]:

$$
\begin{array}{lcl}
\texttt{CAR} & \sqsubseteq & \exists^{=1}\texttt{trans.TRANSMISSION} \sqcap \exists^{=1}\texttt{power.HORSEPOWER} \sqcap ((\leq 1\texttt{airco}) \sqcap \\
 & & \forall\texttt{airco.AIRCONDITIONING} \sqcap (\neg\exists\texttt{airco} \sqcup (\geq_{100} \texttt{power}))
\end{array}
$$

This constraint involves the use of concrete domains and implies the integration of such a domain in the language $\mathcal{FML}$. The integration of concrete domains into DL has been described in [5]. Another constraint, naturally expressed in $\mathcal{FML}$ is the disjointness of features. The fact that the MANUAL and AUTOMATIC feature are disjoint can be expressed as $\texttt{MANUAL} \sqsubseteq \neg\texttt{AUTOMATIC}$.

[1]Note that in first order logic $p \rightarrow q$ is equivalent with $\neg p \vee q$.

[2]$\geq_{100}$ stands for the unary predicate $\{n; n \geq 100\}$of all strict positive integers greater or equal 100.

2.3 Reasoning Tasks in Feature Modeling

Tool support for feature models should at least contain support for the feature notation and the different dependencies and constraints. The use of DL to formalize feature models enables the execution of certain tasks that are now left to the developer. This section shows how standard reasoning tasks of DL can be used to accomplish certain tasks.

The standard reasoning tasks considered in DL at the terminological level are *subsumption, concept consistency* and *knowledge base consistency*. C_2 subsumes C_1 iff in all models of the knowledge base $\mathcal{K}$ the interpretation of C_1 is a subset of the interpretation of C_2. A concept C is consistent in $\mathcal{K}$ if $\mathcal{K}$ admits a model in which C has a non-empty interpretation. A knowledge base $\mathcal{K}$ is consistent if there exists a model for $\mathcal{K}$. $\mathcal{ALCQ}$ is EXPTIME-complete.
Feature model consistency A feature model is consistent if it is possible to implement a system obeying this model. Checking if a model is consistent corresponds to the verification of the feasibility to build a system.
Feature consistency A feature is consistent if it can be instantiated with respect to the feature model. A feature is inconsistent due to, e.g. over-constraining.
Feature subsumption A feature F_2 subsumes a feature F_1 if in all possible instantiations of the feature model the interpretation of F_1 is a subset of the interpretation of F_2. Subsumption gives rise to a classification of all the features appearing in a feature model. It also allows the deduction of properties of one feature from those of another one.
Feature and constraint addition The addition of a feature or constraint can lead to the replacement of some specific subformula within a terminological axiom. This boils down to the addition of a specification. This can be seen as a function θ mapping specifications to specifications. This function is analogous to the δ-function in [4]. The consistency of this addition w.r.t. a knowledge base $\mathcal{K}$ reduces to knowledge base consistency of $\theta(\mathcal{K})$.

In this approach, DL seems to be a natural way to express feature diagrams and some constraints. The inclusion of additional information of a feature model still needs further investigation. Also to which extend the connection between the **GR(K)** language, the multi-modal counterpart of $\mathcal{ALCQ}$ can be useful in this context and the idea of a description language being able to define notions involving self-reference [2].

3 Logic Meta-Programming for Feature Interaction Detection

In the development of a software application, the addition of particular software artifacts (components, aspects, objects, ...), implementing a certain feature, will introduce interactions with other software artifacts. In this approach we try to use a declarative (i.e. logic) metalevel representation of the feature's implementation to detect feature interaction and interference.

3.1 The Figure Editor Case

Consider a simple figure editor in which the user is able to draw points on the screen and interconnect them to form lines and polygons. The basic system's service only allows this functionality. Using an object-oriented language we implement this system according to figure 2. Afterwards, we want to add

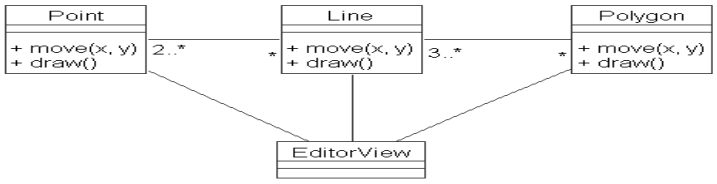


Figure 2: UML Diagram of the Figure Editor.

additional features to the simple figure editor. For instance, we add a feature implementing the archival of figures on a disk and later on, we add a *color* feature allowing to color the points, lines and polygons. After the introduction of the *color* feature, a feature interference can occur between the *archival* and the *color* feature because the original *archival* feature does not store the color of a point, line or polygon.

3.2 Logic Meta-Programming Approach

The Logic Meta-Programming (LMP) technique has an inate capability of declaratively capturing the structure of a program. The metalevel representation of a software application consists of logic facts. A possible representation of our figure editor example is:

```
class(Point).  class(Line).  class(Polygon).
method(Point,move,arguments(x,y),statements(...))
method(Point,draw,arguments([]),statements(...))
...
```

Using logic rules, we can derive a higher-level representation (i.e. towards the design-level). Using such rules, the LMP-technique has been extensively used to detect programming patterns, to trace the impact of changes in the implementation and to check conformance with the corresponding design and architectural description [11], [13].

We now augment the automatically generated metalevel representation of the software program with logic assertions classifying every software artifact in one or more features. For our figure editor example, this means:

```
feature(figures,[class(Point),class(Line),class(Polygon)])
feature(archival,[method(Point,store),method(Line,store),method(Polygon,store)])
feature(figuremovement,[method(Point,move),method(Line,move),method(Polygon,move)])
feature(UI,[class(EditorView),method(Point,draw),method(Line,draw),
          method(Polygon,draw)])
...
```

Because the features are now explicitly defined in terms of the software artifacts that implement them, we are able to reason about the interaction between features using the metalevel representation of the entire program. For each feature we can now determine with which features it interacts directly (through method calls or access of shared variables). In a system implementing a lot of features, a developer could at least derive which features that could be affected by a change in a particular feature (or the addition of a new feature). A logic rule that detects access to the same instance variable by two different features is written as follows[3]:

```
sharedInstanceVariable(?feature1,?feature2,?sharedInstanceVariable) if
    methodInFeature(?feature1,?method1),
    accesses(?method1,?sharedInstanceVariable),
    methodInFeature(?feature2,?method2),
    accesses(?method2,?sharedInstanceVariable).
```

For clarity, we also include the implementation of predicates used in the rule above:

```
methodInFeature(?feature,?methodDescription) if
    feature(?feature,?list), member(method(?className,?methodName),?list),
    methodInClass(?className,?methodName,?methodDescription).

accesses(?method,?instVar) if
     reads(?method,?instVar).
accesses(?method,?instVar) if
     writes(?method,?instVar).
```

The rules `methodInClass`, `reads` and `writes` are part of the SOUL framework developed in [13]. We will not show them here, but they are implemented by several logic rules that reason about the logic metalevel representation.

[3]Logic variables are written using a '?'

However, using these rules, we can only detect feature interactions. To detect feature interference we include logic rules that express constraints or invariants on the implementation. For example:

```
archivalInvariant() if
    classInFeature(figures,?class),
    instVar(?class,?instVar),
    methodInFeature(archival,?method), accesses(?method,?instVar).
```

This rule expresses a simple invariant which states for the *archival* feature that every instance variable in classes of the feature *figures* should be accessed by a method of the *archival* feature. This expresses the condition that the *archival* feature should save every part of the state of the figures.

Adding a new feature to our figure editor might introduce conflicts with the other features, depending on the implementation. As we illustrated, adding colors to the figures will interfere with the *archival* feature. Whether we change the original feature *figures* or we add a complete new *color* feature, the change will boil down to introducing new state variables to `Point`, `Line` and `Polygon` classes. If we do not change the implementation of the `store` method, the archival invariant will not be satisfied and a feature interference will be detected by the resolution engine.

In this experimental approach, LMP promises to be a viable technique to support feature interaction problems in software development. Future research will investigate on a general methodology for feature interaction detection, using LMP.

4 Summary

We described two approaches dealing with feature interactions in software engineering using logic. The first approach defined a formal language for feature modeling in the problem domain using DL. The second approach uses the LMP approach to detect feature interaction in the solution domain. In both approaches, open research questions are related to which kind of information is necessary and sufficient to allow reasoning about feature interactions.

References

[1] R. Accorsi, C. Areces, and M. de Rijke. Towards Feature Interaction via Stable Models. In *Proceedings of the 2nd WFM*, Florianópolis, Brasil, October 1999.

[2] C. Areces. *Logic Engineering. The Case of Description and Hybrid Logics*. PhD thesis, ILLC University of Amsterdam, 2000.

[3] C. Areces, W. Bouma, and M. de Rijke. Description Logics and Feature Interaction. In P. Lambrix, A. Borgida, M. Lenzerini, R. Möller, and P. Patel-Schneider, editors, *Proceedings of the International Workshop on Description Logics (DL'99)*, pages 28–32, 1999.

[4] C. Areces, W. Bouma, and M. de Rijke. Feature Interaction as a Satisfiability Problem. In *Proceedings of* MASCOTS'99, October 1999.

[5] F. Baader and P. Hanschke. A Scheme for Integrating Concrete Domains into Concept Languages. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence, IJCAI-91*, pages 452–457, Sydney (Australia), 1991.

[6] K. Czarnecki and U. W. Eisenecker. *Generative Programming (Methods, Tools, And Applications)*. Addison Wesley, 2000.

[7] Jonathan D. Hay and Joanne M. Atlee. Composing Features and Resolving Interactions. In David S. Rosenblum, editor, *Proceedings of Eighth International Symposium on the Foundations of Software Engineering*, pages 110–119. ACM Press, November 2000.

[8] K. Kang, S, Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute and Carnegie Mellon University, Pittsburgh PA, November 1990.

[9] H. Klaeren, E. Pulvermueller, A. Rashid, and A. Speck. Aspect Composition applying the Design by Contract Principle. In *Proceedings of the Net.ObjectDays2000*, Erfúrt, Germany, October 2000.

[10] Buchheit M., Donini F., and Schaerf A. Decidable Reasoning in Terminological Knowledge Representation Systems. *Journal of Artificial Intelligence Research*, 1:109–138, 1993.

[11] Kim Mens. *Automating Architectural Conformance Checking by means of Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, October 2000.

[12] Keck D. O. and Kuehn P.J. The Feature and Service Interaction Problem in Telecommunications Systems: A Survey. *IEEE Transactions on Software Engineering*, 24(10):779–796, October 1998.

[13] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation.* PhD thesis, Vrije Universiteit Brussel, January 2001.