Using Problem Descriptions to Represent Variability For Context-Aware Applications

Mohammed Salifu Bashar Nuseibeh Lucia Rapanotti Thein Than Tun
The Open University, Milton Keynes, UK
{M.Salifu, B.Nuseibeh, L.Rapanotti, T.T.Tun}@open.ac.uk

Abstract

This paper investigates the potential use of problem descriptions to represent and analyse variability in context-aware software products. By context-aware, we refer to recognition of changes in properties of external domains, which are recognised as affecting the behaviour of products. There are many reasons for changes in the operating environment, from fluctuating resources upon which the product relies, to different operating locations or the presence of objects. There is an increasing expectation for software intensive-devices to be context-aware which, in turn, adds further variability to problem description and analysis. However, we argue in this paper that the capture of contextual variability on current variability representations and analyses has yet to be explored. We illustrate the representation of this type of variability in a pilot study, and conclude with lessons learnt and an agenda for further work.

Keywords

Contextual variability; context-awareness; problem variants; solution variants; product-families

1. Introduction

There is an increasing expectation for software-intensive devices to be context-aware, and many consumer devices such as mobile phones, which are developed as product families, are expected to follow this trend. By context-aware, we mean that products are expected to change their behaviour in response to changes in their operating environments due to changes in properties of domains that are external to them but still affect their behaviour. Reasons for changes in context vary from fluctuating resources upon which a product relies (e.g., reduced bandwidth for a mobile phone) to different operating locations (e.g., a mobile user travelling long distance) or the presence of other objects (e.g., Bluetooth-enabled phones) [7]. Changes may also be caused by users' preferences; for example, users of a mobile phone may require a particular set of features to be available to them while at work and a different set while at home. Mobility is therefore central to our notion of context. This context-induced variability is expected to increase the complexity and scale of traditional variability analysis and management, the impact of which has remained unexamined [23].

The primary objective of our research is to develop an approach to identify, represent, analyse and reason about common and variant sub-problems; and to link such representations to architectural variability types. It is aimed at problem descriptions of product-families operating in varying context environments. Therefore, the emphasis is on contextual variability in the problem space rather than solution space.

Other issues relevant to context-awareness are (1) the monitoring of operating context, (2) the detection of changes in context and (3) the switching of operation from one variant to another. Variation in context may well induce variations in each of these issues.

The remainder of the paper is structured as follows: we begin with a brief overview of related work (section 2), followed by a brief description of problem frames (section 3), which we use as an approach to represent identify problem descriptions. We then describe our proposed approach to represent problem variability (section 4), and provide a detailed illustration of the feasibility and applicability of problem diagrams to describe and reason about problem variations (sections 4.1-4.3). We conclude (section 5) with a discussion of lessons learnt and further work.

2. Related Work

This section briefly discusses current representation of variation points and dependency relations between variants. This will be followed by a discussion of related approaches to context-awareness.

2.1 Variability points and dependencies

Variations in requirements are generally regarded as variations in the intention of a stakeholder in terms of the intended use of an end product [5]. This type of variability has often been modelled and analysed using feature diagrams [34], which capture user-visible functionalities. However, Liaskos et el [23] have observed that variability in requirements may be exacerbated by contextual variability which they refer to as background (or unintentional) variability. They

argue that feature diagrams do not take contextual variability into consideration and are therefore unsuitable for representing and reasoning about variability of systems where contextual changes are common place. Therefore, they have proposed a goal-oriented approach which takes into account both intentional and unintentional variability in its representation.

In an earlier paper [32], we discussed in detail the work of Bachmann and Bass [3] on sources of variability types, and that of Jaring and Bosch [17] on relational dependencies which we argued are consistent with earlier observations by Buhne et el [6]. However, we also looked at other representation using use cases [6]. We concluded that what these approaches have in common is that, none of them explicitly consider the properties of the operating contexts and their constituent domains. For instance, the work of Buhne et el on using use cases to communicate variability to consumers is effective in showing user visible functional dependencies. It is, however, weak at capturing other contextual information such as differences in technology. To the best of our knowledge, Liaskos et el's work is the only attempt at understanding the impact of contextual variability on variant requirement derivation. However, their approach assumes a 'greenfield' development and does explicitly considered the issue of variability in adaptive elements [4]. By adaptive elements, we mean techniques for context monitoring, context change detection and variant switching. These elements are discussed further in sections 4.2 and 4.3.

2.2 Context-Awareness & Adaptive product families

Since our work deals with product-family and context-awareness, we are interested in adaptive approaches that address variability in context and adaptation mechanisms.

Current approaches to context-awareness are largely focused on the use of middleware to support varying contexts [25] such as the One.World approach by [14] and the Odyssey approach by [26] which are aimed at supporting heterogeneity and hiding variations in context from application software. However, Abowd [1] has noted that a middleware-based approach to context-awareness is insufficient, as some contextual changes are only visible in the application level or requires the interpretation of human user. Also, context-aware middleware tends to focus on specific application domains such as hiding variations in platform infrastructure or location details in distributed computing [25]. An example is the work of Apel et el [2] which is aimed at supporting platform heterogeneity in a multi-device varying context environment. However, developing such middleware requires good knowledge of the domain, something not always available.

Current application level approaches to context-awareness also tend to focus on specific problem issues such as self-healing or self-reconfiguring [10] and platform resource fluctuations or differences in the cost of computing resources [31]. Those that are more general, such as the work of Oreizy et el [29], tend to be vague in discussing issues such as monitoring and switching at a very high abstract level, lacking details on what the underlying requirements are. Therefore, the discussion of possible variability in monitoring or switching is absent. In the case of Oreizy et el, an attempt is made to define context-aware infrastructure by a prescribed set of rules with which applications operating in such an environment must comply. But this still does not give sufficient detail as to what the underlying requirements are.

All these approaches are largely solution space oriented to context-aware application development. However, Zhang et el [37] has argued for a requirements approach to adaptive software development, and that the semantics of adaptation is made explicit in requirements. This they noted enables the evaluation of adaptive systems not only in terms of the requirements of problem variants in different context but also in terms of how adaptation is achieved. This position is consistent with Hayes et el [15] arguing for deriving specification of embedded systems from that of its environment. In this case, the specifications are first expressed in terms of the domains of the physical world after which they are derived in terms of the solution machine's interface to the world.

Related work that has tried to deal with both context-awareness and product-families is based on software architecture configuration techniques [2, 12, 13, 19]. Each of these is briefly discussed later in this section. The configuration of an architecture refers to its set of components, their interconnections and the constraints defining the behaviour of this architecture [35]. The replacement of such a configuration with a new (or different) one after it has been released or during the operation of the applications based on it, is referred to as reconfiguration [21, 28].

The work of Gomaa and Hussein [12, 13] is based on the use of architectural styles or patterns, such as the client server architectural style, to construct what they refer to as a reconfiguration pattern (based on the style of the generic architecture). A reconfiguration pattern is used to guide the process of automatically deriving

one product-line member from a different one. This can be argued to be a generalised form of parameterisation [30], as all instances of this product-family must conform to the style and different members are instantiated by changing the values of parameters.

The work of Kim et el. [19] is similar to that of Gomaa and Hussein. The key difference is that Kim et el. provide an architectural description language for describing architectures and modifications to be applied to them during reconfiguration. The example in [19] adopts a pipe-and-filter architectural style and could therefore be argued to be a specialised case of the work of Gomaa and Hussein with the addition of a means to describe the architecture and its modifications.

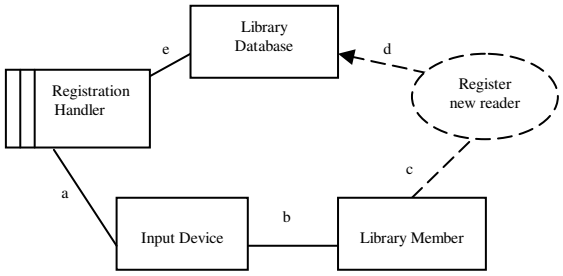
The work of Apel and Bohm [2] is based on the use of a layered architectural style to design a reconfigurable middleware for a context-aware environment. In this work, context-aware environment refers to an operating environment with network bandwidth fluctuations, connection interruptions, device mobility and resource-constrained devices. The services provided by this architecture are operating system-based and largely limited to the network infrastructure. In designing the reconfigurable middleware architecture, Apel et el have adopted the product-line paradigm and produced a generic architecture from which specific architectures tailored to different environments are produced as and when the context requires, during runtime.

Again, to the best of our knowledge, it is only the last three approaches that have considered both product-family and adaptability and in the case of Apel et el context-awareness too. However, these attempts are all solution space oriented and implicitly consider the underlying requirements that lead to the use of their approaches.

3. Problem Frames for Representing Variability

The Problem Frames approach (PF) to requirements engineering provides a conceptual basis for analyzing software problems in context [16]. In this approach, problems comprise three descriptions: (i) a description of the given properties of the world in which the problem resides (domain knowledge), (ii) a description of the required properties of the world (requirement), and (iii) a description of what the machine, or the computer implementing the software, must do to affect the required properties (specification). Unlike other requirements engineering approaches such as Use Cases [8] and Goals [36], PF is particularly suitable for analyzing context-awareness because it emphasizes the need for understanding the physical context of software problems.

We now introduce and discuss briefly some of problem frames notation and concepts relevant to our discussion. This is done with the aid of a simple problem diagram in Figure 1.



b: LM!{Personal Details}, a: ID!{commands}
e: RH!{Reader Record}
d: LD!{New Reader Record}
c: LM!{Personal Details}

Figure 1: A Simple Problem Diagram.

In Figure 1, the rectangles with no stripes (*Library Database*, *Input Device* and *Library Member*) represent physical domains of the problem world whose properties are relevant to the problem. The dashed oval represents the requirement, and the rectangle with a double stripe is the machine domain whose specification is required. Thick lines between the domains present sets of shared properties of the domains involved and are referred to as shared phenomena. For example, the shared phenomenon *e* indicates that details of reader records are shared between the two domains Registration Handler (RH) and Library Database (LD). The prefix RH! suggests that RH can manipulate the reader records, whilst LD can only observe them. The dashed line between the requirement and Library Member (LM) denotes that the requirement references the property of LM, and the dashed line with an arrow head between the requirement and LD denotes that the requirement constrains the property of LD. It means that when the library member provides personal details, a new reader record is expected to be added to the database.

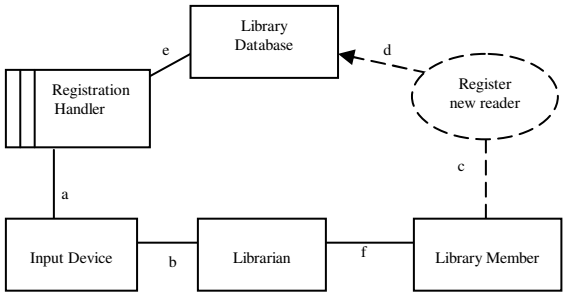
A problem frame is a known class (pattern) of problem with a well understood structure and concern. The problem diagram in Figure 1 represents an instance of a basic type of problem known as the Workpieces frame [16]. The main concern of this frame is as follows:

"A tool is needed to allow a user to create and edit a certain class of computer-processable text or graphic objects, or similar structures, so that they can be subsequently copied, printed, analysed or used in other ways. The problem is to build a machine that can act as this tool."

In Figure 1, the domain *Library Database* is the "computer-processable" object, and *Registration Handler* is the "tool needed" to allow *Library Member* using *Input Device* to "create" a member record.

Although most software problems, when decomposed, are expected to fit the basic frames, Jackson acknowledges that there could be problems that have extra concerns.

A problem variant frame represents a new problem class (pattern) that closely matches a known problem frame such as the workpieces frame but differs because of the presence of a problem domain or control pattern not found in the existing problem frame [16]. One of such variants is called a "connection variant". A connection variant introduces a domain into the basic frame diagram. For example, Figure 2 shows a problem diagram that is similar to the one in Figure 1, with an additional connection domain *Librarian* between *Input Device* and *Library Member*. The new diagram signifies the fact that, rather than library member, it is the librarian who interacts with machine through the input device. Since this problem diagram shares the main concern of the problem diagram in Figure 1, we regard this new problem diagram as a variant of the original problem diagram in Figure 1.



b: L!{Membership Details}, a: ID!{ commands }
e: RH!{Reader Record}
d: LD!{New Reader Record}
c: LM!{Personal Details}
f: LM!{Membership Form}

Figure 2: A Variant Problem Diagram.

In this paper, we use the notion of variant frames to capture contextual variability in context-aware applications.

4. Outline of Our Approach

Given a requirement R, our approach begins with the identification and representation of a problem diagram aiming to fit a known problem frame or a variant of a basic frame. In some cases, R may need to be decomposed into sub-requirements in order to fit known problem frames. Using available knowledge about the problem context, we identify a set of variables (V_1, V_2 … V_m) representing possible sources of contextual variations. Assuming a non-varying context, we construct problem diagram for the requirement. The resulting problem diagram is context-unaware.

We next vary each of the contextual variables one at a time accessing its impact on the context-unaware problem diagram. Where a variation in contextual variable causes requirement R not to be satisfied, we derive a variant problem diagram for this context situation ensuring that R is satisfied. Note that in some cases, it may be necessary to arrange the contextual variables in sequence as they may be some dependency relations between them requiring simultaneous consideration of two or more variables. This case is not considered in this paper but is being explored.

Following the derivation of problem variants for variations in context, we carry out variants analysis and address context-awareness concerns such as the detection of changes in varying context. This involves the identification of domains and phenomena to be monitored in the problem world in order to do so. This may introduce new physical domains inducing new sub-problems into the problem analysis. For instance, problem diagrams to monitor and report changes in physical domains or to update designed lexical domains storing contextual information.

We next consider the composition of problem variants to enable the context-aware product operate in all contexts. Other concerns such as switching, interference, consistency, etc can also be addressed. These concerns are only briefly discussed.

4.1 A pilot study

This study is intended to illustrate the use of problem descriptions for capturing contextual variability in problem variant diagrams. This is done with regards to software applications for context-awareness.

Software is required to control the transmission of pictures from an external digital camera (Concord EyeQ [9]) into a mobile phone's storage (Nokia 9500 [27]) under the control of the phone user. This is to be

done using Bluetooth wireless technology at two different locations. In the first case, the transmission is to be done without encryption while in the second case it should be done using the Secure Socket Layer/Transport Layer Security protocol (SSL/TLS) [18]. These are for secure and non-secure locations respectively. Further details are:

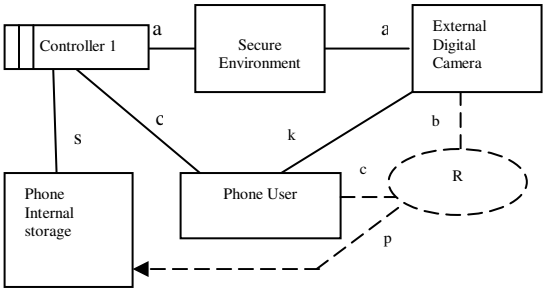
1. The phone makes a request for a picture which must be transmitted, received, and saved on its internal storage.
2. The camera prompts a user when transmission starts and when it stops.
3. All transmissions must be secured. This means that, all picture transmissions in a non-secure location must be encrypted. However, encryption is not necessary in a secure location. The software must adapt its behaviour (i.e. carry out encryption or not) without explicit user involvement.

Considering the pilot study along the three descriptions of problem frames concepts, the following observations are made:

1. The underlying requirement (R) is *a secure transfer of pictures* from a digital camera to the mobile phone's storage.
2. The need to secure or not, represents one source of a contextual variable in W.

The overall requirement (R) fits a problem frame known as Command Behaviour frame. Therefore, there is no need for decomposing R into sub-requirements. Also, assuming a secure location the problem diagram for this is as given in Figure 3. This represents our basic problem diagram and assumes a non-varying context (i.e. all operating locations are secured). Hence, no phenomena relevant to the detection of changes in context are identified and represented.

Using the basic problem diagram in Figure 3 and withdrawing the assumption of a secure location, we realise that the requirement will not be satisfied using this problem diagram in non-secure locations. Therefore, we now derive a variant problem diagram for non-secure locations. To achieve this, we apply a Connection variant as it is suitable for connecting a problem domain to a machine domain where there is a need for intermediate processing. This introduces a domain into the problem diagram to carry out the required encryption/decryption. Figure 4 gives the resulting variant problem diagram.



s:PIS!{receivespicture, savespicture}
a:C1!{RequestTransmission, TerminatesTransmission}
b: EDC!{BeginsTransmission, EndsTransimission}
p:PIS!{receivespicture, savespicture}
c: PU!{StartTransmission, StopTranmission}
k:PU!{ConfirmsStartedTransmission, ConfirmsCompletedTranmission}

Figure 3: A basic problem diagram for secure location.

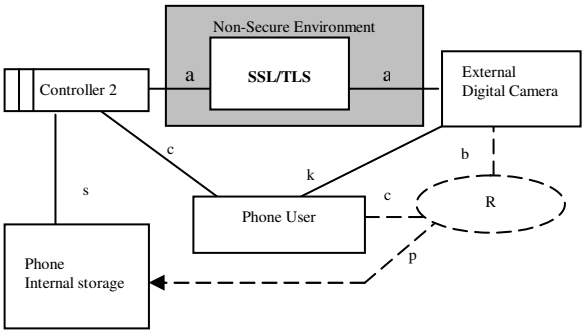


Figure 4: A variant problem diagram for non-secure location.

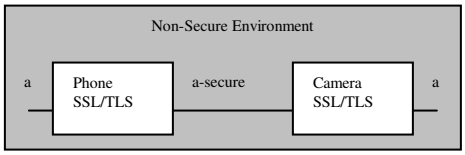


Figure 5: A partial problem diagram of Figure 4 showing further details of the SSL/TLS domain.

Even though the SSL/TLS domain is shown to be a single domain in the problem diagram in Figure 4, a critical look will show otherwise. This is shown in Figure 5.

The introduction of the SSL/TLS domain has resulted in an addition of the shared phenomenon 'a-secure', which is defined as follows:

a-secure: PS!{secRequestTransmission, secTerminatesTransmission}

It is worth noting that SSL/TLS represents a solution to the sub-problem of encryption/decryption which we do not need to solve as the solution is given. If this was not the case, then a sub-problem would have to be introduced to carry out the securing of the transmission channel. Problem frames treats solution machines to sub-problems as given domains when used in other problem diagrams.

The use of problem frames or variant frames in this way contributes to knowledge reuse in the problem space as they represent recurring problem classes. In addition to that however, our introduction of a non-varying context problem diagram from which problem variants are derived for different context situations represents further reuse. This allows us to reuse the analysis done on the original problem in Figure 3 on the one in Figure 4. It also enables reuse in different contexts of the same product. However, non-varying context problem diagrams and their related variants may well be composed and used in different products operating in the same application domain. This is not illustrated in this paper but is being explored.

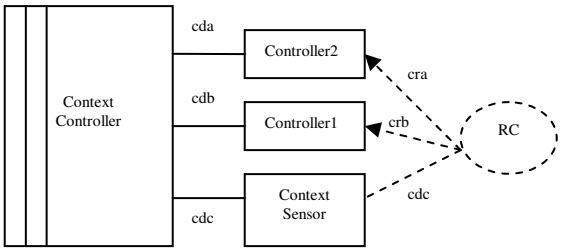
4.2 Problem Variants Analysis

The approach taken in the variant derivation is based on the application of the standard principle of separation of concerns [11]. That is, the concerns of each of the sub-problems were considered independent of that of managing the varying operating context. For instance, the sub-problem shown in Figure 3 is suitable for an operating environment where encryption is not required. The designed machine assumes this fact and will always operate under it. Alternatively, the sub-problem diagram shown in Figure 4 assumes an operating environment where encryption is required and will always operate under this assumption. We therefore consider these two sub-problems as being context-unaware as they assume fixed contexts of operations and therefore do not explicitly carry out checks on the operating environment in order to adjust their behaviour. This approach has enabled us to consider the problems of monitoring, change detection, and managing varying contexts outside the original sub-problem variants. This has been observed by McKinley et el [24] to contribute positively to the development of adaptive software and reuse, both of which are essential to context-awareness.

We now take a closer look at each of the problem variants and attempt to address some of the context-awareness concerns such as monitoring and change detection. Consider a situation in which the transfer of a picture started in a secure location and continues into a non-secure one. There will be a need for suspension of transmission to switch from a non-secure channel to a secure one after which transmission must be resumed at the point for which it was suspended. This is a significant problem, and raises concerns such as initialization and interference [16].

One possible approach to compose the two sub-problems is through the use of composition frames proposed by Laney et el [22]. This effectively inserts a controller between the problem world domains and the sub-machines. All interactions between the sub-machines and the problem world domains are intercepted by the controller and the constraints defined by the composition requirements determine permissible patterns of interactions. In this case, the composition rules are dynamically determined by the environmental properties (secure or non-secure). We define the composition requirement as follows:

RC: In a non-secure location make sure that picture transmission is handled by Controller2. In a secure location make sure that picture transmission is handled by Controller1.



cdc: CS! {SecureEnv, NonSecureEnv}
cda: CC! {Initialize(InitState), Start, Suspend, Enquire(CurrentState)}
cdb: CC! {Initialize(InitState), Start, Suspend, Enquire(CurrentState)}
cra: C2! {Running, Stopped}
crb: C1! {Running, Stopped}

Figure 6: A Composition Problem Diagram.

Figure 6 gives a possible problem diagram for such a composition, referred to as a composition problem diagram. The diagram suggests that the Context

Controller (CC) can monitor the environmental property and switch between Controller1 and Controller2 accordingly to meet RC. In the diagram, CC can enquire current states of Controller1 and Controller2, and initialise Controller1 and Controller2 to required states, as well as start and suspend executing of Controller1 and Controller2. These operations allow CC to switch between secure and non-secure transmission in response to environmental properties dynamically.

4.3 Discussion

Using the problem frames approach allows us to separate composition concerns from the basic problem in each problem variant. We therefore argue that problem frames facilitate separation of concerns by allowing different sections or sub-problems of a bigger problem to be considered individually. They allow for different levels of abstractions of domains and their phenomena in variant problem representations. This, we suggest is useful for the representation of context-aware applications as illustrated in the pilot study.

An alternative approach to that of ours would be to consider the problem of context-awareness in the original problem variants diagrams. That is, Figure s 3 and 4 each address the issue of context-awareness as part of the problem. The problem with this alternative approach in our view is that it (1) tries to address many concerns simultaneously and (2) does not scale as each subsequent addition or derivation of a variant will have to be based on the immediate preceding one (to maximise reuse). These could create difficulties for analysts working on later variants.

5. Conclusions & Further Work

The focus of this paper has been on the representation and analysis of contextual variability due to its importance for context-awareness. We have illustrated, using a pilot study, the use of problem descriptions for capturing contextual variability in problem variant diagrams. This is done using Jackson's notion of variant frames.

Despite the apparent suitability of problem descriptions for representing contextual variability, Jackson's notion of problem variants is restrictive. The current definition requires either the addition of a domain or changes in the control pattern of an existing problem diagram. For instance, the digital camera in our pilot study could easily be replaced with a printer or a projector and a mobile phone could be required to interact with all these in different contexts. In such a case, the replacement of an existing domain with a different domain which in turn will vary the existing phenomena will be required. Therefore, we are seeking to extend the definition of problem variant to take such situations into account. Raising the abstraction level of domains is one possible line of investigation. For instance, raising the abstraction level of the digital camera in the pilot study to a "Bluetooth device" will allow for more variants to be derived. Replacing the camera with a "Bluetooth printer" may result in a Control variant frame.

So far we have considered only one source of contextual variability. Where many sources of contextual variability are present with crosscutting concerns, it may be necessary to use other techniques to capture such variability sources and their dependencies. This will then be used to inform the variant problem derivation and formulation composition requirements. This is currently being investigated.

We have not considered the detail specification of each sub-problem machine and that of the context controller in this paper. This is not necessary in illustrating the fundamentals of our approach. However, in deriving the detail specification of each machine, especially in the case of the context controller machine, it may be necessary to introduce some form of formality in deriving a specification. To this end, we are currently exploring the possible use of event calculus [20, 33] for its suitability in doing so.

Acknowledgements

We thank the EPSRC for their financial support. We also thank our colleagues Andreas Classen, Armstrong Nhlabatsi, Yijun Yu, Robin Laney and Michael Jackson for many useful discussions and feedback on earlier drafts of this paper.

References

1. Abowd, G.D., "Software engineering issues for ubiquitous computing", in ICSE '99, 1999, Los Angeles CA, IEEE CNF.
2. Apel, S. and K. Böhm, "Towards the development of ubiquitous middleware product lines", in Software Engineering and Middleware: 4th International Workshop, SEM 2004, 2005, Linz, Austria.
3. Bachmann, F. and L. Bass, "Managing variability in software architectures", in SSR'01, 2001, Toronto, Ontario, Canada, ACM Press.
4. Berry, D.M., B.H.C. Cheng, and J. Zhang, "The four levels of requirements engineering for and in dynamic adaptive systems", in REFSQ'05, 2005, Porto, Portugal.
5. Bosch, J., "Design & use of software architectures - adopting and evolving a product-line approach", 2000, Great Britain, Addison-Wesley, 1-354.

6. Bühne, S., G. Halmans, and K. Pohl, "Modelling dependencies between variation points in use case diagrams", in Proceedings of the 9th International Workshop on Requirements Engineering Foundation for Software Quality (REFSQ), 2003, Klagenfurt, Austria.
7. Chen, G. and D. Kotz, "A survey of context-aware mobile computing research", in Technical Report TR2000-381, 2000, Dartmouth Computer Science.
8. Cockburn, A., "Writing effective use cases", 1st ed, 2001, Longman, Upper Saddle River, NJ, Addison-Wesley, 1- 304.
9. Geeks.com, C., "Concord eyeq go wireless 2mp bluetooth digital camera", 2006, http://www.geeks.com/details.asp?invtid=EYEQ&cat=CAM. p. 1-2.
10. Georgiadis, I., J. Magee, and J. Kramer, "Self-organising software architectures for distributed systems", in ACM SIGSOFT Workshop on Self-Healing Systems (WOSS '02), 2002, Charleston, South Carolina, ACM.
11. Ghezzi, C., M. Jazayeri, and D. Mandrioli, "Fundamentals of software engineering", Second ed, 2003, Upper Sadle River, New Jersey, Prentice Hall.
12. Gomaa, H. and M. Hussein, "Dynamic software reconfiguration in software product families", Lecture Notes in Computer Science, 2004, 3014/2004, p. 435 - 444.
13. Gomaa, H. and M. Hussein, "Software reconfiguration patterns for dynamic evolution of software architectures", in Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04), 2004, IEEE CNF.
14. Grimm, R., et el., "System support for pervasive applications", ACM Transactions on Computer Systems., 2004, 22(4), p. 421–486.
15. Hayes, I.J., M.A. Jackson, and C.B. Jones, "Determining the specification of a control system from that of its environment", Lecture Notes in Computer Science 2805- Proceedings of FME2003, 2003, p. 154-169.
16. Jackson, M., "Problme frames: Analyzing and structuring software development problems", 1st ed, 2001b, New York, Oxford, Addison-Wesley, 390.
17. Jaring, M. and J. Bosch, "A taxonomy and hierarchy of variability dependencies in software product family engineering", in Proc. of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04), 2004, IEEE CNF.
18. Kegel, D., "Ssl / tls", in Accessed on October 11th 2006, 2001, http://www.kegel.com/ssl/.
19. Kim, M., J. Jeong, and S. Park, "From product lines to self-managed systems: An architecture-based runtime reconfiguration framework", in Workshop on the Design and Evolution of Autonomic Application Software (DEAS 2005, 2005, ACM Press.
20. Kowalski, R. and M. Sergot, "A logic-based calculus of events", New Generation Computing, 1986, 4, p. 67–94.
21. Kramer, J. and J. Magee, "The evolving philosophers problem: Dynamic change management", IEEE Transactions on Software Engineering, 1990, 16(11).
22. Laney, R., et el., "Composing requirements using problem frames", in Proceedings of the 12th International Requirements Engineering Conference (RE'04), 2004, Kyoto Japan., IEEE Computer Society Press.
23. Liaskos, S., et el., "On goal-based variability acquisition and analysis", in 14th IEEE International Requirements Engineering Conference (RE'06), 2006, Minneapolis/St. Paul, Minnesota, USA,, IEEE CNF.
24. Mckinley, P.K., et el., "Composing adaptive software", IEEE Computer, 2004, 37(7), p. 56-64.
25. McKinley, P.K., et el., "Composing adaptive software", Computer, 2004, p. 56-64.
26. Noble, B.D., M. Price, and M. Satyanarayanan, "A programming interface for application-aware adaptation in mobile computing", in February 1995, 1995, School of Computer Science, Carnegie Mellon University. p. 1-14.
27. Nokia, F., "Enterprise: Developing end-to-end systems", 2006, Nokia Forum, Online. p. 1-54.
28. Oreizy, P., et el., "An architecture-based approach to self-adaptive software", Intelligent Systems and Their Applications, IEEE [see also IEEE Intelligent Systems, 1999, 14(3).
29. Oreizy, P., N. Medvidovic, and R.N. Taylor, "Architecture-based runtime software evolution", in Software Engineering, 1998. Proceedings of the 1998 (20th) International Conference on Software Engineering, 1998.
30. Perry, D.E., "Generic architecture descriptions for product lines", in Lecture Notes in Computer Science, 1998, Springer Berlin / Heidelberg.
31. Poladian, V., et el., "Dynamic configuration of resource-aware services", in Proceedings of the 26th International Conference on Software Engineering ICSE '04, 2004, IEEE Computer Society.
32. Salifu, M., B. Nuseibeh, and L. Rapanotti, "Towards context-aware product-family architectures", in First International Workshop on Software Product Management, 2006, Minneapolis, Minosota, US.
33. Shanahan, M., "The event calculus explained." Springer Lecture Notes in Artificial ntelligence, 1999, 1600, p. 409–430.
34. Sochos, P., "Feature models to the architecture", in Proceedings of the First International Software Product Lines Young Researchers Workshop(SPLYR). 2004.
35. van der Hoek, A., "Configurable software architecture in support of configuration management and software deployment", in INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 1999.
36. Van Lamsweerde, A., "Goal-oriented requirements engineering: A guided tour", in Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium, 2002.
37. Zhang, J. and B.H.C. Cheng, "Using temporal logic to specify adaptive program semantics", Architecting Dependable Systems-Journal of Systems and Software (JSS), 2006, 79(10), p. 1361-1369.