Beyond Context-Oriented Software

Kim Mens
Université catholique de Louvain, Belgium
kim.mens@uclouvain.be

Nicolás Cardozo
Universidad de los Andes, Colombia
n.cardozo@uniandes.edu.co

Bruno Dumas
University of Namur, Belgium
bruno.dumas@unamur.be

Anthony Cleve
University of Namur, Belgium
anthony.cleve@unamur.be

Abstract

The last two decades have seen a lot of research on context-aware and context-oriented software development technologies, across subfields of computer science. This research area is slowly starting to mature and researchers currently explore how to unify different solutions proposed in these subfields. We envision that within another decade some of these solutions will make it into mainstream software development approaches, tools and environments. Most end-user software built by that time will be context-aware and able to adapt seamlessly to its context of use (devices, surrounding environment, and users' preferences). This transition from traditional to context-oriented software also requires a mindset shift in users. If users are to accept adaptive systems, they need to be in control. Context-orientation should evolve to become less technology- and more user-centric, putting the user back in control. A first step is to provide good feedback to the user about when and what adaptations take place, and mechanisms to allow users to partly control certain adaptations, followed by easily usable and understandable personalisation mechanisms dedicated to each end user. Eventually, when adaptive systems become completely natural and adopted by end users, this will culminate in our vision where users are in full control of relevant features or adaptations of applications of their interest, selected on-demand from online feature clouds, and integrated automatically into the running system.

Copyright © by the paper's authors. Copying permitted for private and academic purposes.

Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution SATToSE2016 (sattose.org), Bergen, Norway, 11-13 July 2016, published at http://ceur-ws.org

1 A Historical Perspective

The first civil examples of context-aware systems started to appear in the early nineties, with application prototypes that acted as office or personal assistants. Since the turn of the century, the amount of context-aware applications, such as context-aware tour guides [38], increased rapidly, by taking advantage of smart mobile devices, sensors or smart objects. Such smart devices are able to perceive their context of execution, communicate with each other, and interact with people [29]. With these new found capabilities, software systems have become increasingly sensitive to their context. In contrast to traditional software, the behaviour of context-aware systems depends not only on their input and output, but also on their context of use, including the time, place, weather, user preferences, system internals, and interaction habits [26].

In 2003, Rohn [38] predicted the first context-aware systems to become commercially available by 2007 and to reach maturity by 2035. Today, many applications on smart devices indeed exhibit context-aware features. Different programming paradigms to develop context-aware systems have been proposed over the last two decades [39, 40]. Such new trends to make software systems more aware of and adaptive to context have even started to appear in mainstream languages, in the form of application frameworks like OSGi[1] or ReactiveBlocks.[2] We expect the ideas and mechanisms of context-aware systems to slowly percolate and become more available and adopted over the following decade, when most end-user applications will be developed with dedicated context-oriented technology so that software systems can seamlessly adapt to their users and context of use. By that time, the fact that software knows its context of use and how it needs to adapt, will have become a feature expected by every user.

This ongoing trend towards context-oriented software development has opened many questions from a technological perspective as well as from a human perspective. From a computer science perspective, the problem of building context-oriented software systems that can adapt dynamically to changing contexts has been studied from at least three different angles, each proposing independent solutions to manage adaptations in their own domain. In the domain of software development, programming language research has explored novel programming paradigms to dynamically adapt the behaviour of running systems according to detected context changes [22, 40]. In the domain of information systems, database research has studied context-aware database technology and more flexible query languages [5, 16, 28, 31]. In the domain of human-computer interaction (HCI), research has studied the problem from the point of view of user interfaces [3, 8, 27], including multimodal interfaces [18, 19], adapting software systems' interfaces to particular devices or (group of) users.

From a human perspective, the notion of context in computer science as a user-centered concept can be seen as an individualising paradigm, raising interesting ethical and sociological questions [4, 20]. Nevertheless, despite the fact that the concept of "context" is a key issue in humanities and social sciences, these areas have not yet researched in detail the notion of context in computing, and the impact it has on how users interact with a software system, or with their environment through a software system.

Today, these different research perspectives are still largely disconnected and independent, which begs for more unified approaches that reconcile the progress in these different domains. Current research [30] is starting to look at how to integrate these proposals of context-orientation in different domains into a single unified approach. We expect that quite some progress towards such unified approaches will be achieved in the next decade. Yet, something is still missing for this paradigm to become truly mature and accepted: a user-centric vision.

2 Putting the user in control

Context-orientation can be seen as a new modularisation mechanism that supports the trend towards ever more dynamicity and context-awareness. As stated in the previous section, in the coming years we expect context-oriented technology to reach some level of maturity and adoption.

[1]https://www.osgi.org

[2]http://www.bitreactive.com

But what will still be missing for this technology to reach full maturity, is for the technology to become "really" user-centric by putting the end-user in control. So far, software development technology focused mainly on systems that are developed *for* users, and not *by* users. The fine-grained dynamic composition and conflict resolution mechanisms, offered by context-oriented software development technology, provides an ideal basis to finally put the power of building software in the hands of the end user. The user will become able to incrementally build or adapt applications, even as they are being executed, by selecting or deselecting the fine-grained features chosen by users, from online feature clouds [14], pushing the idea of features-as-a-service to the extreme. Dependencies and incompatibilities between features would be taken into account by the composition mechanism. Selected features would be customized to the user's habits and typical contexts of use, integrating them automatically into the running system. We foresee four important steps (in the following order) that need to be taken to put the user in control:

1. **Self-explanation:** If end-users are ever to accept highly adaptive systems, they need to feel they remain in control of what is happening with their system. Therefore, whenever the software adapts its behaviour in response to context changes, the system should provide clues to the user about what was adapted and why [25], yet without being too intrusive by asking the user for its input or reaction with every adaptation.

2. **Active customization:** In case users do not accept adaptations, it is necessary to provide mechanisms for users to reject adaptations, gaining actual control over the behaviour of their system. Defining how users can "undo" or "redo" adaptations triggered by context changes remains to be explored.

3. **Maleable tools:** To regain control of the system, end users interact with the system and their environment via a toolset. Such tools need to be easy to manipulate and understand, for regular users to control their system customisations or customisation policies.

4. **Context-oriented software:** Once adaptive systems finally find widespread adoption and become natural to end users, fully user-tailored systems will be a reality, providing software services that, at run time, are composed autonomously from a set of available fine-grained features in feature clouds.

3 User-centric context-oriented software

Before addressing the challenges to be faced in achieving our vision of feature clouds, this section briefly identifies the key characteristics of user-centric context-oriented software.

Dynamic Adaptation

The software should be able to adapt, during its execution, without requiring any (or as little as possible) explicit user intervention, to new and sometimes even unexpected situations, and to the appearance, disappearance or modification of features discovered in available feature clouds. Composition and adaptation of appearing and disappearing features in the feature cloud should react to context changes autonomously. Autonomous behaviour adaptation to the context and to new entities can be managed at the programming level using COP and service adaptivity [13, 11]. Such ideas could also be explored to manage the adaptation of user interface and database objects.

Context Awareness

Software systems should be aware of their current execution context and surrounding environment (including the user) in order to exhibit the most appropriate behaviour according to the situation at hand.

Context Orientation

More strongly, software systems should have an explicit representation of the current context of use [21], to reason about it, and dynamically adapt their behaviour appropriately to the current context.

User Centricity

The software should not only carry the notion of context at its heart, but should also have an explicit representation of its users, so as to be able to best adapt to users' preferences, desires, and habits. In particular, user information could be regarded as a subset of the "context of use" to which the software should adapt. This appears as an independent characteristic, given that it is crucial for users to perceive a deep level of customization to their preferences, for them to adopt and feel in control of the technology.

Predictability

The technology used to achieve such dynamic adaptation to context should provide sufficient guarantees that the software keeps on exhibiting predictable behaviour according to the initial expectancies of the system, even when unforeseen changes occur. Different approaches to assure the consistency of behaviour adaptations have been proposed. The majority of these approaches present a formal run-time model that manages newly introduced behaviour, verifying/validating that such behavior complies with the formal model [15, 12, 23, 34].

Resilience

Related to the previous characteristic, the software should be sufficiently robust and resilient to unexpected changes, so that it keeps on functioning, though perhaps with reduced functionality (*i.e.,* applying self-healing techniques [36], for example by resorting to some default behaviour), as opposed to crashing or aborting with an error.

User feedback and control

A software system that is continuously undergoing changes at run time may quickly become overwhelming to end users, who may get the feeling they are no longer in control of the system. To avoid user discomfort due to the ever-changing behaviour of the system it is desirable to provide users with enough feedback about the decisions taken by the system, so that they regain control over the system.

Automation

Adaptation should be automated, in the sense that it should not require explicit user intervention (or only a very minimum). Interactions with the user should be informative or to request preference settings in the light of new adaptations.

Non-intrusiveness

Adaptations to the system should be non-intrusive, not hindering users in the tasks they are currently executing, even for those cases where the software needs to inform the user about important changes.

Scalability

The software technology should be scalable in terms of number of contexts, features, users, or collaborating devices that can be handled. Well-designed modularisation, composition, scoping and verification mechanisms need to be foreseen to achieve such scalability.

Habitability

The technology should not be too complex and should be designed carefully, and with the necessary tool and development support, so that developers and users feel “at home” with the technology and “at ease” to build or compose such systems.

Verifiability

Related to the properties of resilience and predictability, and given the high evolutivity of context-oriented systems, there is a strong need for formalisms and techniques to verify relevant properties of the system, whether it be during analysis and design time, or at run time. If at run time, again, the verification and its effects should be as non-intrusive as possible.

Integration

All characteristics above should be integrated at different technological and process levels in a unified way. At the technological level, the behavioural, user interface, and database aspects of context-awareness in the system should be reconciled. This, by realising the concepts of context awareness at each level with a unified API, so that the integration of all three levels is smooth; as opposed to current solutions were no such integration exists. Similarly, at the process level, requirements, design models and source code should be unified, to ease the integration of the different domains, while still focusing on each of their individual concerns. For example, the use of extended variability models with notions of context awareness could be an entry point for unifying context awareness and adaptation concepts present in all three perspectives, to build next generation software systems. (1) In software development, variability models can be mapped to specific programming language constructs defining and managing behaviour adaptations [17]. (2) In information systems, variability models can be used to introduce data adaptations, without changing the data definition models [1]. (3) In HCI, such models could be used to define the different adaptation possibilities of user interfaces [9].

User-tailorability

Finally, and maybe even most importantly, to achieve the vision presented in this paper, the software should be user-tailorable at run time, allowing users to select and deselect which software features they would like to include or exclude in *their* software system, even while the system is running.

Note that the characteristics described above for our vision of feature clouds touch upon similar characteristics targeted by other kinds of systems that manage and offer software services aware of their context. A typical example of such systems is that of service mashups. Mashups are introduced to integrate data from multiple sources and provide extended services from said data sources’ combination. Mashups proved useful as visualisation services, by combining geo-location information with some other service’s data. For example, a common visualisation mashup is to display posts for a social media outlet in a map. This can be used to measure traffic and use trends for a particular area during a particular period of time. Furthermore, mashups can use sensor data to specialize provided services to the current situation in the surrounding environment [6]. Mashup definitions are sequential, following a defined structure in which data can be combined. While the idea behind mashups is to offer services where the user is the main concern, mashups’ sequential definition makes them difficult to control by end users. This is due to the rigidity of such definitions, where to reject an adaptation, users would need to reject the full mashup. Even more, mashups are proposed as tools for developers, managing mashups may be too complex for end users. Dynamic mashup solutions are explored to tackle the rigidity problem. Dynamic mashup definitions [41] introduce service types as a means to group services providing similar functionality and abstract the interaction with concrete mashup services. This model also permits the dynamic selection of mashups according to the surrounding environment. Unfortunately, the definition of mashups remains largely targeted to developers and technical users; regular end users would struggle to control the mashups faced with the complexity of their programming language encoding. Moreover, such definition takes place at development time for all possible execution environments. Users do not have the possibility to tailor their system for particular environments accepting different sets of adaptations in each one.

Recent approaches propose the idea of service composition using an autonomous process where fine-grained service composition is learnt from their interaction in a particular environment [11]. This proposal follows feature clouds' ideas of composing fine-grained behaviour definitions from a cloud of features. Nonetheless, the proposal is fully automated, leaving end users out of the loop, which may compromise the approach's acceptance.

4 Challenges

Many of the characteristics identified in the previous section have been or are being explored in several areas of software engineering; in particular in the fields of context-oriented programming [40, 12, 2, 13], context-aware databases [32, 33, 16], and adaptive user interfaces [19, 37]. In this section we highlight some harder challenges in the area of context-awareness, related to the unification of the concepts in each of these specific fields. However, many other important challenges remain to be addressed as well. Issues concerning distribution [13], or security and privacy [20] are equally important for achieving full acceptance of adaptive technology to context. Nonetheless, this paper focuses more on how to address, in a unified way, the technological aspects to enable context-awareness from different perspectives. Concepts of distribution, privacy and security can be built later over these core concepts.

A Unified Approach

It is not easy to strike a right balance in achieving *all* characteristics presented in Section 3 within a single unified approach, since many of them have competing goals. Different fields may prefer different solutions or trade-offs towards achieving these characteristics. Reconciling all these competing views and solutions requires a truly multi-disciplinary approach within and across fields. We now present an overview of competing characteristics in the development of user-centric context-aware systems, which is summarised in Table 1. Characteristics marked as (⇒⇐) represent that a middle ground must be found between the two characteristics, whereas characteristics marked as (⇐⇒) represent a conflicting objective between the characteristics, and only one of them can be attained successfully. Each of the trade-offs is evaluated with respect to one of the perspectives to develop adaptive systems.

Characteristics		**COP**	**Data bases**	**User interfaces**
Dynamic adaptation	Predictability	⇐⇒		
Dynamic adaptation	User centricity			⇐⇒
Dynamic adaptation	User feedback & control	⇒⇐		⇒⇐
Context awareness	User centricity		⇒⇐	⇒⇐
Verifiability	Scalability	⇒⇐	⇒⇐	⇒⇐
Automation	User feedback & control		⇐⇒	⇐⇒
Integration	Habitability	⇒⇐	⇒⇐	⇒⇐

Table 1: Characteristics trade-offs.

Dynamic adaptation vs. Predictability. Feature clouds posit an adaptation model enabling software systems to adapt any part of their functionality at any moment in time, using fine-grained features. Under this model, it becomes difficult to foresee how the system will behave at run time [7]. To predict all possible adaptations at all possible execution times, all possible program traces would need to be known beforehand. This would turn systems too rigid, questioning their adaptivity.

Dynamic adaptation vs. User centricity. Adapting user interfaces is also at odds with User centricity, since adaptations modify the way users interact with the system. If some functionality is not visible anymore, or it changes its appearance, users may be unsure about what to expect from the system [35].

Dynamic adaptation vs. User feedback & control. Similar to the previous trade-off, adaptation of user interfaces gives users the impression the system is incomprehensible and not under their control [35]. Giving users too much decision power over adaptations would render the system too rigid, hampering the dynamicity of adaptations, as the system would require users to take an active action for every possible adaptation. Similarly, informing the user about every adaptation would hamper systems' dynamicity. Moreover, users could still feel they are not in control of the system, given that all adaptations would take place without their knowledge. A balance needs to be found between the feedback given to users [25], and the actions they can take over adaptations (*e.g.,* rejecting or accepting and adaptation), without presenting these options for all adaptations, but just some of them. Defining and deciding which adaptation are users' responsibility, which are seamless to users, and which require informing users is an open research topic.

Context awareness vs. User centricity. While users should be able to tailor the system to their liking, this should not go to the extreme where the system is no longer taking into account the context but just the users' preferences configuration. A balance should be found between the personalization attributes that can be inferred from the context, and those that should be set as user preferences.

Verifiability vs. Scalability. The objective with feature clouds is to have different environments in which a multitude of adaptations defined as fine-grained features are deployed to interact with each other and be composed with other applications roaming around the environment. To be successful, many adaptations would need to be available for each environment. If not done carefully, behaviour introduced or removed by adaptations may break the correct functioning of the system. To avoid situations in which the system behaviour may break, it is possible to run a verification algorithm assuring adapted behaviour preserves system correctness. However, as the amount of adaptations in the system grows, the time required for its verification increases. If too much time is spent in the verification phase, the system may become unusable. If the system has too many adaptations its verification might be unfeasible, to increase the scale of the systems we can build, it is required to define verification techniques from the three development perspectives. These techniques should be incremental and be integrated with one another to exploit as much as possible the results obtained previously to accelerate the verification process.

Automation vs. User feedback & control. Adaptations should take place based on the surrounding context and user preferences without further user intervention. Interactions with the user should be restricted to provide or request information about adaptation preferences. To achieve this, there are two seemingly diverging paths. On the one hand, changes should happen as seamlessly as possible. On the other hand, users should remain informed of important changes to the system's behaviour and should have the possibility to reject such changes. A non disruptive way to provide information to users, and an interesting research direction, is to use ambient output designed to take advantage of the users' background processing capabilities [24].

Integration vs. Habitability. The three perspectives to enable software adaptations provide their unique models and definitions of what an adaptation is, how it is defined, and how it should take place in the system. These perspectives do not necessarily couple with each other. When unifying the three perspectives, special attention must be paid, not to over complicate the model so that users are no longer able to build and compose their desired systems.

Scalability

Whereas scalability in terms of the number of contexts, users, or devices to be handled may remain manageable (since these are often restricted by the scope of the application), the vision of having online feature clouds containing a myriad of fine-grained features may pose a major scalability issue. The problem of combining features selected from a feature cloud is acute, since features are not necessarily different, some features may provide behaviour similar to others, while the differentiation between them is key to the system's quality of service. Moreover, each feature is applicable to certain contexts. Problems such as: (1) how to assure all (and only) relevant features are discovered by users, (2) how to develop features that manage dependencies with unforeseen features, and (3) how to handle incompatibilities between features, become even more problematic with the system's envisioned granularity and scale.

Automated

Having an automated feature composition mechanism also poses a major challenge (apart from the scalability challenge presented above). Indeed, not only features that were designed to work together can be combined, but the composition with all features in the cloud could be possible. To some extent, the composition mechanism will do its best to combine any features users find relevant to combine, even when this was not anticipated. This could occur often given the potential size and dynamicity of feature clouds, which makes it impossible to foresee all possible combinations, as some features will only appear in the environment long after the deployment of others.

Best effort

Given the competing goals of high dynamicity, adaptability, and context-awareness on the one hand, versus guaranteeing predictability, resilience and robustness on the other, it may not always be possible to compose the desired combination of features. In such cases, the composition mechanism needs to resort to a best-effort approach to propose a composition that is robust while remaining as close as possible to the user's desires. Deciding what is the 'best' solution is challenging as it may depend on the context and goals of the application as well as on the user's perception.

Acceptable

Finally, probably the most important challenge to be addressed for the technology to mature, will be to create adaptive systems that users can understand, accept, and ultimately adopt. This touches upon many of the aforementioned characteristics and challenges such as user centricity, predictability, resilience, non-intrusiveness, user feedback, automation, scalability and user-tailorability. Empirical studies as well as test scenarios (*e.g.,* A/B testing) are required to assess whether these systems are desirable or acceptable by users and in what form.

5 The Road Ahead

To achieve our user-centric vision of context-aware feature clouds, we need an advanced feature selection and software composition solution where:

1. At a high-level, the features and the contexts they depend upon, are presented to the end-user in an *easy-to-understand* way, that clearly depicts the intra- and inter-relationships between features and contexts (for example, using a context feature model [10]).

2. Users can then select, on-the-fly, the high-level features they desire from this feature cloud. The relations present in the model, combined with a recommendation system based on past decisions, may be helpful for the user to choose the most appropriate features.

3. Once chosen, the different features selected are combined automatically according to the context.

4. This high-level model already allows the verification of some consistency properties of the proposed feature cloud. Potential composition conflicts are resolved dynamically according

to pre-defined composition policies, managed by an underlying lower-level composition mechanism.

5. The lower-level composition mechanism is similar to current-day context-oriented programming approaches, which define features as fine-grained building blocks that describe small pieces of functionality relevant to particular contexts.

6. These primitive building blocks can be combined into larger ones through composition policies. In addition to default predefined composition policies, the programmer can define customized policies, that may depend on the context. Even more, to some extent the composition policies can be tailored by end users, for example, to declare what conflict resolution rules they prefer under what circumstances.

7. At composition time, taking into account the composition policies and the relationships between contexts and features, a further verification of the consistency of the composition can be performed.

8. Acceptability studies at user interaction level will help define guidelines that will channel the composition policies and necessary mechanisms at the user interface level.

Observe how in this solution the line between end-users and developers starts to blur, since both can define combinations of features, albeit at a different level. The developer mostly declares low-level features and combinations thereof, to be offered to the end-user. End-users select sets of features they would like to see combined, and the system then composes them automatically based on the high-level relationships between these features (and contexts) and the low-level composition policies declared by developers (some of which may be fine-tuned by end-users themselves).

6 Summary

With the advent of ubiquitous and mobile computing and the Internet of Things, software systems are more and more required to adapt to their surrounding and execution environments. As a consequence, research on context-oriented software development has seen multiple achievements, in different domains, during the last two decades. However, although impressive research advances have been made, context-aware aspects of software systems often remain hard-coded. We believe this is due to the dispersion of research results over multiple domains within and beyond computer science, and that a multidisciplinary approach with the user as focal point is needed to progress further. In this paper, we advocated a vision of context-oriented software built from fine-grained features gathered from online feature clouds. We presented a four-step plan towards a user-centric approach of context-oriented systems. We then identified a number of characteristics that define user-centric context-oriented software. Whereas some of these characteristics have been explored by researchers, many challenges still lie in front of us if we are to achieve true end-user and software developer acceptance and satisfaction. We finally proposed to address these challenges with an advanced feature selection and software composition solution that blurs the line between software developers and users.

References

[1] L. Abo Zaid and O. De Troyer. Towards modeling data variability in software product lines. In T. Halpin, S. Nurcan, J. Krogstie, P. Soffer, E. Proper, R. Schmidt, and I. Bider, editors, *Enterprise, Business-Process and Information Systems Modeling*, volume 81 of *Lecture Notes in Business Information Processing*, pages 453–467, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[2] T. Aotani, T. Kamina, and H. Masuhara. Type-safe layer-introduced base functions with imperative layer activation. In *Proceedings of the 7th International Workshop on Context-Oriented Programming*, COP'15, pages 8:1–8:7, New York, NY, USA, 2015. ACM.

[3] D. Billsus, C. A. Brunk, C. Evans, B. Gladish, and M. Pazzani. Adaptive interfaces for ubiquitous web access. *Commun. ACM*, 45(5):34–38, May 2002.

[4] J. Bohn, V. Coroamă, M. Langheinrich, F. Mattern, and M. Rohs. Social, economic, and ethical implications of ambient intelligence and ubiquitous computing. In *Ambient intelligence*, pages 5–29. Springer, 2005.

[5] C. Bolchini, C. A. Curino, G. Orsi, E. Quintarelli, R. Rossato, F. A. Schreiber, and L. Tanca. And what can context do for data? *Commun. ACM*, 52(11):136–140, Nov. 2009.

[6] A. Brodt, D. Nicklas, S. Sathish, and B. Mitschang. Context-aware mashups for mobile devices. In *Proceedings of the 9th international conference on Web Information Systems Engineering*, WISE'08, pages 280–291, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola. Self-adaptive software needs quantitative verification at runtime. *Communications of the ACM*, 55(9):69–77, Sept. 2012.

[8] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, and J. Vanderdonckt. A unifying reference framework for multi-target user interfaces. *Interacting with Computers*, 15(3):289–308, 2003.

[9] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, N. Souchon, L. Bouillon, M. Florins, J. Vanderdonckt, and J. V. Plasticity of user interfaces: A revisited reference framework. In *In Task Models and Diagrams for User Interface Design*, pages 127–134. Publishing House, 2002.

[10] R. Capilla, O. Ortiz, and M. Hinchey. Context variability for context-aware systems. *Computer*, 47(2):85–87, Feb 2014.

[11] N. Cardozo. Emergent software services. In *In Proceedings of the ACM International Symposium on New Ideas and Reflections on Software*, Onward'16, New York, NY, USA, October 2016. ACM.

[12] N. Cardozo, L. Christophe, C. De Roover, and W. De Meuter. Run-time validation of behavioral adaptations. In *Proceedings of 6th International Workshop on Context-Oriented Programming*, COP'14, pages 5:1–5:6, New York, NY, USA, 2014. ACM.

[13] N. Cardozo and S. Clarke. Context slices: Lightweight discovery of behavioral adaptations. In *Proceedings of the Context-Oriented Programming Workshop*, COP'15, pages 2:1–2:6. ACM, July 2015.

[14] N. Cardozo, W. De Meuter, K. Mens, S. González, and P.-Y. Orban. Features on demand. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS '14, pages 18:1–18:8, Sophia Antipolis, France, 2013. ACM.

[15] N. Cardozo, S. González, K. Mens, R. V. D. Straeten, J. Vallejos, and T. D'Hondt. Semantics for consistent activation in context-oriented systems. *Information and Software Technology*, 58:71 – 94, 2015.

[16] S. Castro, S. González, K. Mens, and M. Denker. Dynamicschema: a lightweight persistency framework for context-oriented data management. In *Proceedings of the International Workshop on Context-Oriented Programming (COP 2012)*, pages 5:1–5:6. ACM, 2012.

[17] B. Desmet, J. Vallejos, P. Costanza, W. De Meuter, and T. D'Hondt. Context-orientet domain analysis. In *Modeling and Using Context, Sixth International and Interdisciplinary Conference on Modeling and Using Context*, pages 178–191, August 2007.

[18] C. Duarte and L. Carriço. A conceptual framework for developing adaptive multimodal applications. In *Proceedings of the 11th International Conference on Intelligent User Interfaces*, IUI '06, pages 132–139, Sydney, Australia, 2006. ACM.

[19] B. Dumas, M. Solórzano, and B. Signer. Design guidelines for adaptive multimodal mobile input solutions. In *Proceedings of MobileHCI'13*, pages 285–294, Munich, Germany, 2013. ACM.

[20] M. Friedewald, E. Vildjiounaite, Y. Punie, and D. Wright. The brave new world of ambient intelligence: An analysis of scenarios regarding privacy, identity and security issues. In *Security in Pervasive Computing*, pages 119–133. Springer, 2006.

[21] S. González, N. Cardozo, K. Mens, A. Cádiz, J.-C. Libbrecht, and J. Goffaux. Subjective-c: Bringing context to mobile platform programming. In B. Malloy, S. Staab, and M. van den Brand, editors, *Proceedings of the International Conference on Proceedings of the International Conference on Software Language Engineering*, volume 6563 of *series-lncs*, pages 246 – 265. Springer, June 2010.

[22] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, March-April 2008.

[23] M. U. Iftikhar and D. Weyns. Activforms: Active formal models for self-adaptation. In *Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS'14. ACM, June 2014.

[24] H. Ishii, C. Wisneski, S. Brave, A. Dahley, M. Gorbet, B. Ullmer, and P. Yarin. ambientroom: Integrating ambient media with architectural space. In *CHI 98 Conference Summary on Human Factors in Computing Systems*, CHI '98, pages 173–174, Los Angeles, California, USA, 1998. ACM.

[25] K. Leichtenstern, E. André, and E. Kurdyukova. Managing user trust for self-adaptive ubiquitous computing systems. In *Proceedings of the 8th International Conference on Advances in Mobile Computing and Multimedia*, MoMM'10, pages 409–414, New York, NY, USA, 2010. ACM.

[26] H. Lieberman and T. Selker. Out of context: Computer systems that adapt to, and learn from, context. *IBM Systems Journal*, 39(3&4):617–631, 2000.

[27] U. Malinowski, T. Kühme, H. Dieterich, and M. Schneider-Hufschmidt. A taxonomy of adaptive user interfaces. In *Proceedings of the Conference on People and Computers VII*, HCI'92, pages 391–414, York, United Kingdom, 1993. Cambridge University Press.

[28] D. Martinenghi and R. Torlone. A logical approach to context-aware databases. In *Management of the Interconnected World*, pages 211–219. Physica-Verlag HD, 2010.

[29] F. Mattern and C. Floerkemeier. From the internet of computers to the internet of things. *Informatik-Spektrum*, 33(2):107–121, 2010.

[30] K. Mens, N. Cardozo, B. Dumas, and A. Cleve. Breaking the walls: A unified vision on context-oriented software engineering, 2015. 14th BElgian-NEtherlands software eVOLution seminar - BENEVOL.

[31] M. Mori and A. Cleve. Feature-based adaptation of database schemas. In *Proc. of MOMPES 2012*, volume 7706 of *Lecture Notes in Computer Science*, pages 85–105. Springer, 2013.

[32] M. Mori and A. Cleve. Towards highly adaptive data-intensive systems: A research agenda. In *Proceedings of First International Workshop on Variability Support in Information Systems (VarIS 2013)*, volume 148 of *Lecture Notes in Business Information Processing*, pages 386–401. Springer, 2013.

[33] M. Mori, A. Cleve, and P. Inverardi. A stability-aware approach to continuous self-adaptation of data-intensive systems. In *Proceedings of the 2nd International Conference on Context-Aware Systems and Applications (ICCASA 2013)*. Springer Verlag, 2013.

[34] L. Nahabedian, V. Braberman, N. D'Ippolito, S. Honiden, J. Kramer, K. Tei, and S. Uchitel. Assured and correct dynamic update of controllers. In *Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS'16, pages 96–107, New York, NY, USA, 2016. ACM.

[35] T. F. Paymans, J. Lindenberg, and M. Neerincx. Usability trade-offs for adaptive user interfaces: Ease of use and learnability. In *Proc. of IUI'04*, pages 301–303, Funchal, Madeira, Portugal, 2004. ACM.

[36] H. Psaier and S. Dustdar. A survey on self-healing systems: approaches and systems. *Computing*, 91(1):43–73, August 2010.

[37] S. Rafiqi, S. Nair, and E. Fernandez. Cognitive and context-aware applications. In *Proceedings of the 7th International Conference on PErvasive Technologies Related to Assistive Environments*, PETRA '14, pages 23:1–23:7, New York, NY, USA, 2014. ACM.

[38] E. Rohn. Predicting context aware computing performance. *Ubiquity*, 2003(February):1–17, Feb. 2003.

[39] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2):14:1–14:42, May 2009.

[40] G. Salvaneschi, C. Ghezzi, and M. Pradella. Context-oriented programming: A software engineering perspective. *Journal of Systems and Software*, 85(8):1801–1817, August 2012.

[41] J. Vallejos, J. Huang, P. Costanza, W. De Meuter, and T. D'Hondt. A programming language approach for context-aware mash-ups. In *Third International Workshop on Web APIs and Services Mashups*, Mashups'09. ACM, October 2009.