Managing the Context Interaction Problem

A Classification and Design Space of Conflict Resolution Techniques in Dynamically Adaptive Software Systems

Kim Mens
Benoît Duhoux
ICTEAM, Université catholique de Louvain, Belgium
[kim.mens,benoit.duhoux]@uclouvain.be

Nicolás Cardozo
Systems and Computing Engineering Department,
Universidad de los Andes, Colombia
n.cardozo@uniandes.edu.co

ABSTRACT

The *context interaction problem* occurs in dynamically adaptive software systems whenever adaptations to different contexts provide incompatible behaviour, as they were not foreseen to occur simultaneously. Several strategies have been proposed to resolve such conflicts when they occur. This paper surveys a number of such conflict resolution strategies, and proposes a design space in which to classify, compare, and explain the differences between them.

CCS CONCEPTS

•General and reference →Surveys and overviews; •Software and its engineering →*Multiparadigm languages;* Software design tradeoffs;

KEYWORDS

context interaction problem, dynamically adaptive software systems, conflict resolution techniques

ACM Reference format:
Kim Mens, Benoît Duhoux, and Nicolás Cardozo. 2017. Managing the Context Interaction Problem. In *Proceedings of Programming '17, Brussels, Belgium, April 03-06, 2017,* 6 pages.
DOI: http://dx.doi.org/10.1145/3079368.3079385

1 INTRODUCTION

Context-aware, context-oriented and dynamically adaptive software systems all share the common property that they adapt their behaviour in different ways to a variety of different situations, gathered from their surrounding execution environment [32]. As a consequence, they face the problem of subtle conflicts that may arise when the system needs to react in different incompatible ways to combinations of situations that were not foreseen. We refer to this problem as the *context interaction problem*, inspired by similar interaction problems in other variability approaches, such as the aspect interaction [30] and feature interaction problems [2, 9].

Programming '17, Brussels, Belgium

978-1-4503-4836-2/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3079368.3079385

To illustrate the problem, consider the example of a home automation system with an emergency response module that reacts to detected emergency situations. Whereas all emergency situations have a common process to follow (alerting the user), the specific steps needed to respond to a particular emergency vary according to each specific situation (*e.g.,* a fire, burglary, or water leak). This module can thus be developed as a dynamically adaptive system, which adapts the response process dynamically to the detected situation. For example, in case a water leak is detected, the system may respond by temporarily turning off the main water supply; in case of a burglary it alerts the police department and turns on the alarm; and in case of fire it activates the sprinklers and the alarm, alerting the fire brigade. When designing such systems, developers should try to consider all possible situations and their combinations up front, since unforeseen combinations of behavioural adaptations may lead to subtle errors. For example, if a fire and a water leak were not foreseen to take place simultaneously, when they do occur together it may happen that the system shuts off the water supply, causing the sprinklers not to work and the house to burn down.

In practice, however, it is not always feasible nor possible to foresee all such conflicts, given the potentially large number of combinations of situations that may occur and their corresponding behavioural variations. Nevertheless, dynamically adaptive systems should be resilient to such conflicting context-specific behavioural adaptations. Therefore, a variety of mechanisms have been studied to detect such context interaction conflicts, and to take appropriate resolution actions when they occur. In the example above, upon detection of the conflict the system could choose not to turn off the water supply so that the sprinklers can still be used. Even though this may cause water damage, the house would not burn down.

In this paper, we explore different strategies that have been proposed to resolve conflicts arising from the unexpected interaction between incompatible context-specific behavioural adaptations. We propose a design space in which to compare these techniques, provide a classification of conflict resolution strategies, and relate them to that design space. The purpose of this paper is to serve as a frame of reference for developers of dedicated context-aware, dynamically adaptive architectures, languages and frameworks, providing them with a better idea of what resolution mechanisms exist to address the context interaction problem.

2 DESIGN SPACE

2.1 Methodology

This section sketches our multi-dimensional design space used to classify different resolution strategies for managing conflicts arising

Venue	2006		2007		2008		2009		2010		2011		2012		2013		2014		2015		2016	
	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R
TAAS	8	0	14	3	19	1	21	2	14	0	26	2	33	5	19	1	20	1	28	3	22	2
SEAMS	8	3	17	3	17	1	17	3	13	2	26	5	21	4	19	4	18	4	18	7	17	4
SASO			25	2	42	3	30	3	25	2	21	0	26	0	34	5	19	1	18	2	20	1
COP							10	5	6	2	7	5	5	2	6	2	8	4	8	2	6	3

Table 1: Landscape of conflict resolution strategies in the area of context-aware and dynamically adaptive systems.

from (unforeseen) interactions between adaptations. Taking inspiration from various resolution strategies collected from research literature, we organised the design space according to five dimensions, corresponding to five key questions (what/where, when, who, how and why) regarding the resolution strategies.

To build a comprehensive design space we conducted a literature review of resolution strategies for managing conflicting context-aware behaviour. The review focused on context-aware, context-oriented, and adaptive systems, by considering publications from top conferences and workshops (SEAMS,[1] SASO,[2] and COP[3]) and journals (TAAS[4]) in these areas. As summarised in Table 1, for each of these venues we collected all publications concerning conflict resolution strategies (R) amongst all published papers at the venue (P). Papers were identified and chosen based on the paper's abstract and content overview, to verify whether the paper indeed presents a strategy to deal with conflicts between adaptations.

From Table 1, we can observe that the bulk of papers describing conflict resolution strategies comes from the SEAMS and COP communities, as these have focused more on the software engineering aspects of dynamic software adaptation. Papers presented at SASO and TAAS tend to have a broader scope, which explains their lower ratio of conflict resolution papers. Selected references for the different conflict resolution strategies are provided in Section 3, where we present each specific strategy as part of our classification.

2.2 The Five Dimensions to Conflict Resolution

Figure 1 depicts our design space, structured along five dimensions, each corresponding to one of five "W" questions (**W**hat/**W**here, **W**hen, **W**ho, **W**hy and ho**W**). Moreover, in this design space, more intrusive approaches can be found closer to the origin, than strategies that are less intrusive for users. We will refer to this design space as the "design space for concern interaction management techniques", since it can easily be generalised to other concern interaction problems than the context interaction problem.

***WHAT / WHERE** is the root cause of the conflict?* Dynamically adaptive software systems are constructed out of different kinds of entities such as **sensors** (detecting situations to which the system should adapt), **contexts** (reifying these situations) and **adaptations** (defining the behavioural adaptation to be executed in a particular situation). As context interaction conflicts occur, knowing what kinds of entities are causing the underlying conflict may influence the most appropriate resolution strategy to use. Conflicts caused by defective sensors that produce data that is absurd,

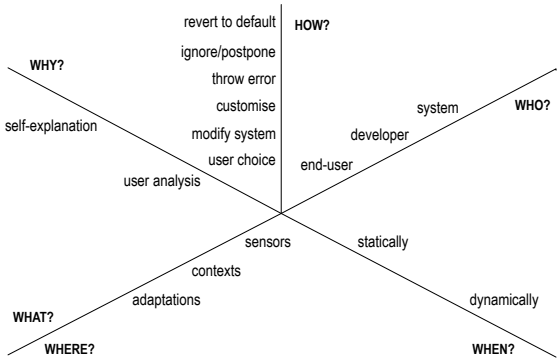

Figure 1: Design space for concern interaction management techniques

unlikely or inconsistent with respect to other sensors could be resolved by simply ignoring the adaptation associated to the situation detected by the sensor, given that the sensor is faulty. Another strategy could be to consult other sources of information to confirm, refute, or correct information provided by the faulty sensor. Conflicts may also be caused by the simultaneous occurrence of two contexts not expected to occur together (due to an incomplete specification of context interactions in the system design). In such cases, a possible strategy may be to ignore the adaptation corresponding to one of the contexts. Which of the contexts to ignore would depend on the specific system domain. Related to the previous conflict, even when contexts were foreseen to interact, the adaptations associated with such contexts could be conflicting. Two different adaptations could adapt a single system feature in different ways according to their associated context. Each of the adaptations is correct in isolation, but their simultaneous execution raises an ambiguity about which adapted behaviour to execute. A linearisation strategy that dictates their combination and execution order could avoid such ambiguity.

***WHEN** does a conflict occur? **WHEN** should it be resolved?* A second dimension along which to classify different conflict resolution strategies is related to the timeliness of the strategy. Conflicts could be resolved either **statically** (*i.e.,* at design or configuration time) or **dynamically** (*i.e.,* during system execution), requiring different conflict resolution strategies. For example, statically, we may detect that two adaptations have been defined for two specific contexts, but no adaptation was defined for their combined context. A simple static resolution could be to require the definition of an additional variant for the combined case. Detecting all such situations statically may not be feasible due to the dynamicity and

[1] http://seams2016.jgreen.de/?page_id=41
[2] http://www.saso-conference.org
[3] http://2017.ecoop.org/series/COP
[4] http://taas.acm.org

combinatorial explosion of context interactions. Missing adaptations for specific context combinations will then be detected only when the situations occur. In such cases, a possible resolution could be to use the behaviour of only one of the adaptations based on some prioritisation strategy.

***WHO** should resolve a conflict?* A third dimension to distinguish conflict resolution strategies is who is responsible for the resolution. Resolution could either be delegated to the system's **end-user** (*e.g.,* asking users which adaptation they prefer to use), be specified by **developers** using dedicated mechanisms (*e.g.,* language constructs or annotations) to define a resolution strategy (*e.g.,* explicitly declared priorities for conflicting adaptations), or be resolved automatically by the underlying **system** or **language** (*e.g.,* an implicit linearisation mechanism in case of ambiguities).

***HOW** to resolve conflicts?* The fourth dimension addresses how to resolve conflicts, which may depend on the other dimensions. For example, resolving conflicts dynamically requires different techniques (*e.g.,* incremental analysis) than resolving them statically (*e.g.,* static analysis) (**WHEN**). Moreover, identified conflicts can be resolved either by end-users or autonomously by the system (**WHO**), and at the level of sensors, contexts, or adaptations (**WHAT/WHERE**). Specific techniques vary between: asking end-users to **choose** what adaptations they prefer; requiring designers or developers to **modify the system**, **customise** a resolution strategy, **throw** a runtime exception, letting the system **ignore** or **postpone** adaptations, or to **revert** to a default behaviour.

***WHY** do conflicts occur?* The final dimension concerns the reasons behind conflicts. Taking into account **WHAT** caused a conflict and its specific resolution (**WHEN, WHO, HOW**), it is possible to provide some explanation of the conflict. This explanation, for example, in case of ambiguity between two incompatible adaptations, can help developers to **analyse** what specific resolution strategy would be most appropriate (*e.g.,* defining a new adaptation, choosing one of the two adaptations, reverting to a default adaptation, or ignoring the adaptation altogether) based on domain knowledge about the system. This analysis is useful to know how to resolve similar types of conflicts in the future. Related to this, automated **self-explanation** techniques [35] can be applied to provide feedback about the resolution strategy used in face of a conflict, and the reason behind using such strategy.

3 CLASSIFICATION

This section presents a classification of different conflict resolution strategies, while linking them back to the design space. Before doing so, we briefly reiterate a typical context interaction problem that such techniques need to address. A dynamically adaptive software system, such as our home automation system, can dynamically adapt part of its behaviour, such as how it responds to emergencies, to different contexts. The default behaviour could be to simply alert the user in case of an emergency. The default behaviour is adapted whenever a fire or water leak is detected. A context interaction problem arises when both situations (fire and water leak) occur simultaneously, so that the system does not know to which of the contexts to respond or how. A variety of different conflict resolution techniques have been proposed to address such problems:

Using Runtime Exceptions. A direct strategy to "resolve" context interaction problems is to let the system (**WHO/system**) automatically raise an exception (**HOW/throw error**) whenever the conflicting contexts occur simultaneously during the system's execution (**WHEN/dynamically**) [26].

Time of Adaptation. Interacting adaptations can cause conflicts in the observable system behaviour by requiring behaviour that is no longer available due to a context change, or because required behaviour is shadowed by an incoming adaptation. It is thus of utmost importance to precisely define when adaptations are incorporated to or removed from the system. Adaptations can be adopted loyally, promptly, or as a combination of both [11].

A *loyal strategy* [5, 17] consists in preventing the inclusion of adaptations associated with a newly active context, until the currently executing context exits. The strategy is thus loyal to the currently executing behaviour, in the sense that it finishes all pending behaviour executions in its current context, before switching to the new context. A loyal strategy (**WHO/system**) avoids conflicts at the level of **contexts** (**WHAT**), ensuring that all entities affected by adaptations associated to a given context finish executing in the same context they started their execution by **ignoring** (**HOW**) the incorporation of new contexts as they are signaled for activation (**WHEN/dynamically**).

A *prompt strategy* [21] tries to adopt adaptations as soon as their associated context is sensed, regardless of the currently executing behaviour. This strategy autonomously addresses conflicts (**WHO/system**; **WHEN/dynamically**) between contexts by giving **priority** (**HOW**) to the latest sensed **context** (**WHAT**), with the purpose of always executing the behaviour most specific to the current situation. However, it may cause the system's observable behaviour to become unpredictable as the adaptation order depends on the last context detected. In our running example, depending on what emergency was detected last, either the house will burn down, or the house will be flooded.

A middle ground between the loyal and prompt strategies is the *prompt-loyal strategy* [11, 33]. This strategy applies adaptations to all components not currently executing, and restrains new adaptations (**HOW/postpone**) coming from **context** changes (**WHAT**) until the behaviour affected by the adaptation has finished its execution (**WHEN/dynamically**). As soon as the behaviour execution finishes, the adaptation is adopted by the system. In this way, the **system** (**WHO**) remains loyal to the current execution, while still promptly adopting new contexts.

Reverting to Default Behaviour. Whenever two adaptations are simultaneously applicable, a conflict may arise when both compete to adapt the same behaviour. Given that the **system** (**WHO**) does not know which of the two adaptations is most appropriate, a conservative resolution strategy is to simply **revert** (**HOW**) to a default behaviour, generic to both adaptations [18, 20, 29, 33]. This strategy subsumes the loyal strategy, since it ignores the conflicting adaptations altogether, even if no other variant of the behaviour is currently executing. The idea behind this conservative strategy is to avoid the conflict by falling back to a default case, at the cost of loosing the specialised behaviour provided by the applicable adaptations. Note that this strategy does assume that a default

behaviour is defined for all possible adaptation combinations. If not, remaining conflicts may still occur.

In practice, this strategy is less conservative than it seems, since several adaptation techniques [15, 18, 24] keep track of active contexts, reverting as a default case to the first less specific adaptation common to both conflicting adaptations, as opposed to reverting to a base behaviour not specialised to any context. Nevertheless, even if this strategy makes a lot of sense in many cases, it goes a bit against the spirit of dynamic adaptation, by conservatively reverting to more generic cases as soon as conflicts arise. Furthermore, the default case may not always be the most appropriate strategy. In our emergency response module, reverting to default behaviour would imply that users are only alerted of an emergency, but neither the fire nor the water leak adaptation would be activated, leading to the house to burn and flood, which in our example is undesired.

Prioritising Adaptation Execution. Given the conflict previously described between **adaptations** (**WHAT**), a resolution strategy is to avoid ambiguities by prioritising the adaptations involved in the conflict. This prioritisation could either be defined by developers, for example using explicit annotations, or be imposed by the language, using an implicit linearisation strategy or activation policy. In our example, such a prioritisation could be used to express that the adaption for fire takes priority over that for water leaks.

Explicit priorities (**HOW/modify system**) defined by **developers** (**WHO**) [19], remove ambiguity between adaptations by considering the adaptation with the highest priority the most specific (**WHEN/statically**). However, if two adaptations with the same priority are applicable to a behaviour, a conflict still remains.

Instead of relying on developer annotations, some programming languages or run-time architectures (**WHO/system**) define a prioritisation scheme by imposing a well-defined linearisation or activation policy (**HOW/customise**). The Context Traits [22] COP language, for example, defines a prioritisation policy based on the activation age of contexts. That is, the most specific adaptations correspond to contexts that are activated more recently, overriding adaptations corresponding to older contexts. The intuition behind this policy is that adaptation associated to more recent contexts are more appropriate to the situations sensed, than adaptations associated to earlier context changes.

User-driven. Another strategy to resolve ambiguity in case of conflicting adaptations is to let **users** (**WHO**) define which of the adaptations to choose. Gathering a user's choice (**HOW**) can be done either by requesting explicit input from the user [14], using a previous analysis of available adaptations [34], or by learning information from observing the user's current activities or past behaviour. In each of these cases, we still need to decide how to use the gathered information, and who is responsible for resolving the conflict based on that information and how.

Requesting explicit user input (**WHEN/dynamically**), whenever adaptations are to take place, allows application users to **choose** (**HOW**) the adaptation they consider most appropriate. Relying on a previous analysis of the system (**WHEN/statically**), on the other hand, can be used to identify possible conflicts between adaptations through debugging and automated analyses. Using the resulting analysis information, **developers** (**WHO**) can then **modify the system** (**HOW**) to remove the interaction problem (*e.g.,* by defining priorities or ensuring a composition order for the adaptations). Finally, the **system** (**WHO**) can derive information by observing previous user actions (**WHEN/dynamically**) to infer (**HOW/customise**) what is their preferred behaviour, for different contexts and combinations of adaptations. The best approach to collect user information about their adaptation preferences may vary and depend on the specific application domain and objectives of the system (*e.g.,* if a fully ubiquitous system is desired).

User as a Context. Related to the previous strategy, an interesting resolution strategy is to regard the users themselves as a **context** [8, 10] (**WHAT**). The idea behind this strategy is to fall back on existing context-aware adaptation mechanisms to let the **system** (**WHO**) adapt its behaviour **dynamically** (**WHEN**) based on the user's preferences and behaviour (**HOW**). Although conceptually this is a very nice approach, it does not work for all possible conflicts. For example, it is very hard to pinpoint what the user context or attribute would be to discriminate between the fire adaptation and the water leak adaptation. However, if we have additional information on the users' preferences and intentions [28], context-conflict management strategies can taking into account those intentions, to maximise users' satisfaction based on their declared preferences. For example, a user could indicate that she is more sensitive to damage caused by fire rather than by a water leak, so that the conflict resolution mechanism can take this sensitivity into account and give preference to the adaptation that turns on the sprinklers.

Formal Conflict Verification. Conflict resolution strategies can also rely on formal approaches, backed up by a mathematical formalism, to verify the coherence of the system upon dynamic adaptations to the context [13]. The formalism of COntext Petri Nets [12, 15], for example, allows to model statically a system's **contexts** and their relationships (**WHAT**), while assuring, upon context activation (**WHEN/dynamically**), that the model is respected. If this is not the case, the system reverts to a previous consistent state (**HOW/ignored**). Similar approaches have been defined upon alternative formalisms such as labelled transitions system [23].

Taking advantage of the type system specification [1, 6, 24], it is possible to verify that no needed behaviour will be missing whenever a **context** (**WHAT**) change occurs. During the system's definition (**WHEN/statically**), a type inference process (**WHO/system**) can identify execution traces that would yield an error due to part of the expected behaviour being missing (**HOW/throw error**). Similarly, the type system definition (**WHO/developer**) can also be used to define the way in which **adaptations** (**WHAT**) should interact with each other to avoid conflicts as the system goes from one adaptation to the next [16] (**HOW/revert to default**).

Orchestration. Interactions between adaptations can also be managed by the **system** (**WHO**), using a control structure assuring that conflicts will not take place in the system. This structure controlling the adaptations, normally a graph-based representation of the interactions between adaptations, is created **dynamically** (**WHEN**) as adaptations are defined or appear in the environment.

Programmatically, it is possible to define (**WHO/developer**) explicit transitions between different **contexts** (**WHAT**) [25], generating a network of allowed interactions between contexts. If

a transition between contexts is not defined or is explicitly forbidden, the adaptations associated with such context are **ignored** (**HOW**). Similarly, based on the defined contexts and adaptations, a graph controlling the transition between contexts can be generated (**WHO/system**) [4, 7, 15]. Finally, adaptations can be orchestrated using feedback loops [3] managing their interaction. The advantage of using such loops is that it is possible to build analysis and verification mechanisms for inter- and intra- loop conflicts. Identification of such conflicts (**WHEN/statically**), can support the design and implementation of the system by **modifying the system** (**HOW**) beforehand, to avoid the conflicts altogether.

Adaptation contracts. Related to formal strategies, it is possible to **statically** define adaptation contracts between adaptation modules [27, 31], specifying the way they should interact with each other, avoiding conflicts **dynamically** (**WHEN**). Adaptation contracts are realized at two different times, at system definition (specifying the contracts), and their dynamic validation (*i.e.,* their enforcement). Therefore, this strategy uses a mixture of dimensions from the design space. Contracts are extracted by **developers** (**WHO**) from the system requirements, and defined in a programming language using a formal (*e.g.,* logic) specification (**HOW/customise**). Even though such contracts are defined statically by developers, the expressive power of the formal specification enables developers to express interaction rules even with future adaptations that are not yet present in the system. Contracts are enforced dynamically by the system (**WHO**), enabling only those adaptations that satisfy the given contracts, while **ignoring** other adaptations (**HOW**).

4 DISCUSSION

In the previous sections we introduced the context interaction problem, and presented a design space as well as a classification of different conflict resolution strategies to address that problem. In this section, we take a step back to discuss in more detail some remaining issues.

Exhaustiveness. A first relevant question is whether our design space and classification are exhaustive. Surely other relevant dimensions along which to classify conflict resolution strategies can be found. However, we purposefully limited ourselves to five dimensions corresponding to five key questions to be asked about any strategy, given that we wanted a design space that is sufficiently generic and easy to understand and remember.

Our partitioning of each dimension into subcategories was based on the strategies we surveyed and how we felt they would fit into each dimension. This partitioning may evolve further as we add more strategies to our classification. This is especially the case for the **HOW** dimension, which surely should be refined further. Our current partitioning of dimensions into subcategories is not always ideal either, even for seemingly obvious subcategories such as **statically** or **dynamically** in the **WHEN** dimension. Indeed, some techniques (like adaptation contracts or some of the formal verification techniques) clearly have both a static and a dynamic counterpart. They require a developer to specify statically what is allowed, while verifying dynamically whether this specification is respected, and taking appropriate resolution actions if not. For the **WHO** dimension too, a more fine-grained division may be needed, for example to distinguish between programmers and designers.

Generality. Whereas the classification in this paper only covers conflict *resolution* strategies, and not techniques for *detecting* those conflicts, we do believe that our design space would be equally suited to classify conflict detection techniques as well, provided a revision of the partitioning of each dimension in subcategories.

Similarly, the design space we proposed seems sufficiently generic so that it could be applied to concern interaction management techniques other than mere context interaction (e.g., feature interaction, aspect interaction, component interaction).

Combining strategies. Section 3 classified and highlighted different conflict resolution strategies in isolation. In practice, however, when building a full dynamically adaptive system, a single strategy may not suffice to resolve all possible conflicts. Suppose, for example, that we are using prioritisation as resolution strategy for disambiguating conflicts. This strategy is able to resolve conflicts, except in cases where different contexts or adaptations are defined with the *same* priority. In such cases an alternative backup strategy would be needed. It is thus necessary to further explore *how* different individual strategies can be combined into a coherent unified resolution strategy that performs best in most cases. Additionally, it is important to evaluate *what* impact such combined strategies would have on the overall system behaviour.

Customisation. Combining different strategies is, however, not sufficient to handle all possible interaction conflicts, since in some cases the most appropriate solution depends on the particular contexts and adaptations involved. In addition to the combination of different strategies, there is thus a need for customising existing strategies to particular cases. It should also be assessed at what level such customisation is most appropriate (*e.g.,* customisation by the user, by the developer or learned by the system).

Intrusiveness. A trade-off exists between customisability and intrusiveness. On the one hand, customised strategies are likely to be more appropriate since they have been targeted to solve a particular conflict type. On the other hand, defining such customised strategies is often more intrusive since it requires either a developer to override a more generic strategy, or to gather explicit user feedback giving information on how to handle that conflict. A less disruptive strategy would be to infer user preferences based on previous user interactions with the system. This strategy, however, requires the implementation of a learning component as part of the system.

User acceptance. On the one hand, end-users of dynamically adaptive systems would like their applications to be as adaptive as possible, so that they provide personalised services to the current context for every user. On the other hand, end-users may not accept systems that are constantly asking for input. Flooding users with interactions is too disruptive, since every possible adaptation interrupts the user's activity and stalls the system until it receives a response. This violates the objective of context-aware systems as ubiquitous systems. Again, systems that learn from past behaviour may be a possible solution to this problem. Nonetheless, such systems should give sufficient explanation to their users about when, how, and why adaptations take place. Providing feedback

about decisions taken by the system can mitigate users' resistance to systems that perform actions on their behalf.

Context of use. It is important to note that some strategies may make more sense in some usage contexts than in others. A typical example is the strategy of throwing a runtime exception. Whereas this strategy is clearly to be avoided in deployed end-user applications, it may be a valid strategy to use during system development, to catch as many problematic cases as possible before system deployment. Once the system is deployed, however, this quite invasive strategy should be replaced by a more conservative strategy such as reverting to default behaviour.

Meta strategies. Given the previous observation, there may be a need for meta strategies, *i.e.,* strategies that determine, based on the context of use, what is the most appropriate strategy for that context. As such, the chosen strategy is too a contextual adaptation.

5 CONCLUSION

This paper provided an overview of current resolution strategies for the context interaction problem. The objective of the paper is to serve as a reference framework for developers of context-aware systems, to be better informed about the kind of problems that may arise due to the unforeseen interaction between contexts and adaptations, and about possible strategies for resolving these conflicts as well as the trade-offs these strategies face. While we do not claim our classification of resolution strategies to be exhaustive, our design space does cover the key questions when faced with a conflict. Since the proposed design space maps specific types of conflicts to resolution strategies, it can be of use for the development of context-aware systems, or languages or frameworks for managing such systems. The design space can easily be generalised to serve as a framework to reason about other kinds of concern interaction management techniques as well.

REFERENCES

[1] T. Aotani, T. Kamina, and H. Masuhara. 2015. Type-Safe Layer-Introduced Base Functions with Imperative Layer Activation. In *Proc. of the 7th Workshop on Context-Oriented Programming (COP'15)*. ACM, 8:1–8:7.

[2] S. Apel, J. M. Atlee, L. Baresi, and P. Zave. 2014. Feature Interactions: The Next Generation (Dagstuhl Seminar 14281). *Dagstuhl Reports* 4 (2014), 1–24.

[3] P. Arcaini, E. Riccobene, and P. Scandurra. 2015. Modeling and Analyzing MAPE-K Feedback Loops for Self-Adaptation. In *Symp on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'15)*. IEEE/ACM, 13–23.

[4] E. Bainomugisha, W. De Meuter, and T. D'Hondt. 2009. Towards Context-Aware Propagators: Language Constructs for Context-Aware Adaptation Dependencies. In *Proc. ofWorkshop on Context-Oriented Programming (COP'09)*. ACM, 8:1–8:4.

[5] E. Bainomugisha, J. Vallejos, C. De Roover, A. Lombide Carreton, and W. De Meuter. 2012. Interruptible Context-dependent Executions: A Fresh Look at Programming Context-aware Applications. In *Proc. of the ACM Symp on New Ideas and Reflections on Software (OnWard'12)*. ACM, 67–84.

[6] D. Basile, L. Galletta, and G. Mezzetti. 2015. Safe Adaptation Through Implicit Effect Coercion. In *Essays Dedicated to Pier Paolo Degano on Programming Languages with Applications to Biology and Security*. Springer-Verlag, 122–141.

[7] N. Bencomo, A. Belaggoun, and V. Issarny. 2013. Dynamic Decision Networks for Decision-making in Self-adaptive Systems: A Case Study. In *Proc. of the Symp on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'13)*. IEEE, 113–122.

[8] H. Eon Byun and K. Cheverst. 2001. Exploiting user models and context-awaR.ss to support personal daily activities. In *Workshop on User Modeling for Context-Aware Applications (UM2001)*.

[9] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. 2003. Feature Interaction: A Critical Review and Considered Forecast. *Computer Networks* 41 (2003), 115–141.

[10] J. Cámara, G. A. Moreno, and D. Garlan. 2015. Reasoning about Human Participation in Self-Adaptive Systems. In *Symp on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'15)*. 146–156.

[11] N. Cardozo, S. González, K. Mens, and T. D'Hondt. 2011. Safer Context (de)Activation: Through the Prompt-Loyal Strategy. In *Workshop on Context-Oriented Programming (COP'11)*. ACM, 1 – 6.

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

[13] N. Cardozo, K. Mens, and S. Clarke. 2017. Models for the Consistent Interaction of Adaptations in Self-Adaptive Systems. In *Software Engineering for Self-Adaptive Systems: Assurances*. LNCS, Vol. 9640. Springer-Verlag. To be published.

[14] N. Cardozo, K. Mens, S. González, P-Y. Orban, and W. De Meuter. 2014. Features on Demand. In *Proc. of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS'14)*. ACM, 18:1–18:8.

[15] N. Cardozo, J. Vallejos, S. González, K. Mens, and T. D'Hondt. 2012. Context Petri Nets: Enabling Consistent Composition of Context-Dependent Behavior. In *Workshop on Petri Nets and Software Engineering (PNSE'12)*, Vol. 851. CEUR Workshop Proceedings, 156 – 170.

[16] D. Clarke, P. Costanza, and ft. Tanter. 2009. How should context-escaping closures proceed. In *Workshop on Context-Oriented Programming (COP'09)*. ACM, 1–6.

[17] B. Desmet, K. Vanhaesebrouck, J. Vallejos, P. Costanza, and W. De Meuter. 2007. The Puzzle Approach for designing Context-Enabled Applications. In *Conf. of the Chilean Computer Science Society*. IEEE, 23 – 29.

[18] D. Ghosh, R. Sharman, H. R. Rao, and S. Upadhyaya. 2007. Self-healing systems — survey and synthesis. *Decision Support Systems* 42, 4 (2007), 2164 – 2185.

[19] S. González, N. Cardozo, K. Mens, A. Cádiz, J-C. Librecht, and J. Goffaux. 2010. Subjective-C: Bringing Context to Mobile Platform Programming. In *Proc. of the Conf. on Software Language Engineering (LNCS)*, Vol. 6563. Springer, 246 – 265.

[20] S. González, M. Denker, and K. Mens. 2009. Transactional Contexts harnessing the Power of Context-Oriented Reflection. In *Workshop on Context-Oriented Programming (COP'09)*. ACM.

[21] S. González, K. Mens, and A. Cádiz. 2008. Context-Oriented Programming with the Ambient Object System. *Jour. of Universal Computer Science* 14, 20 (2008), 3307–3332.

[22] S. González, K. Mens, M. Colacioiu, and W. Cazzola. 2013. Context Traits: Dynamic Behaviour Adaptation Through Run-time Trait Recomposition. In *Proc. of the Conf. on Aspect-oriented Software Development (AOSD'13)*. ACM, 209–220.

[23] M. Usman Iftikhar and Danny Weyns. 2014. ActivFORMS: Active Formal Models for Self-adaptation. In *Proc. of the Symp on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'14)*. ACM, New York, NY, USA, 125–134.

[24] H. Inoue, A. Igarashi, M. Appeltauer, and R. Hirschfeld. 2014. Towards Type-Safe JCop: A Type System for Layer Inheritance and First-class Layers. In *Proc. of the Workshop on Context-Oriented Programming (COP'14)*. ACM, 7:1–7:6.

[25] T. Kamina, T. Aotani, and H. Masuhara. 2011. EventCJ A Context-Oriented Programming Language with Declarative Event-based Context Transition. In *Conf. on Aspect Oriented Software Development (AOSD'11)*. ACM, 253–264.

[26] T. Kamina and T. Tamai. 2009. Towards Safe and Flexible Object Adaptation. In *Workshop on Context-Oriented Programming (COP'09)*. ACM.

[27] M. Mongiello, P. Pelliccione, and M. Sciancalepore. 2015. AC-Contract: Run-Time Verification of Context-Aware Applications. In *Symp. on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'15)*. 24–34.

[28] I Park, D. Lee, and S. J. Hyun. 2005. A dynamic context-conflict management scheme for group-aware ubiquitous computing environments. In *29th Annual International Computer Software and Applications Conference (COMPSAC'05)*. 359–364 Vol. 2.

[29] H. Psaier and S. Dustdar. 2011. A survey on self-healing systems: approaches and systems. *Computing* 91, 1 (January 2011), 43–73.

[30] F. Sanen, E. Truyen, W. Joosen, A. Jackson, A. Nedos, S. Clarke, N. Loughran, and A. Rashid. 2006. Classifying and documenting aspect interactions. In *Proc. of the AOSD workshop on aspects, components, and patterns for infrastructure software (Technical Report CS–2006–01)*. University of Virginia Computer Science, 23–26.

[31] A. Sartorio and M. Cristiá. 2009. First Approximation to DHD Design and Implementation. *CLEI electronic Jour.* 12, 1 (2009).

[32] F. Schreiber and E. Panigati. 2014. Context-Aware Software Approaches: a Comparison and an Integration Proposal. In *27th Italian Symp on Advanced Database Systems (SEBD'14)*. 175–184.

[33] N. Taing, M. Wutzler, T. Springer, N. Cardozo, and A. Schill. 2016. Consistent Unanticipated Adaptation for Context-Dependent Applications. In *Workshop on Context-Oriented Programming (COP'16)*. ACM, 1–6.

[34] S. Uchio, N. Ubayashi, and Y. Kamei. 2011. CJAdviser: SMT-based Debugging Support for ContextJ. In *Proc. of the Workshop on Context-Oriented Programming (COP '11)*. ACM, 7:1–7:6.

[35] K. Welsh, N. Bencomo, P. Sawyer, and J. Whittle. 2012. Self-Explanation in Adaptive Systems. In *Conf. on Engineering of Complex Computer Systems (ICECCS'12)*. 157–166.