Multi-dimensional Dependency and Conflict Resolution for Self-adaptable Context-aware Systems

Davy Preuveneers
Department of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A, Leuven, Belgium
davy.preuveneers@cs.kuleuven.be

Yolande Berbers
Department of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A, Leuven, Belgium
yolande.berbers@cs.kuleuven.be

Abstract

Self-adaptable systems dynamically adapt to satisfy new functionality requirements, to optimize performance and to adapt to variable runtime conditions. Context-aware systems acquire and utilize information about the context of a user and a device to provide personalized applications. Adaptation in a transparent way for the user is an important concern for both systems. To decrease the adaptation complexity of self-adaptable context-aware systems, we propose to model self-adaptation across three layers of abstractions: the runtime infrastructure, the context middleware and the user applications. The aim is to achieve per layer self-adaptation of the composition or behavior of these layers without jeopardizing the integrity and usability of the overall system. We discuss a multi-dimensional dependency and conflict resolution mechanism to guide self-adaptation and find a stable reconfiguration of the system at runtime.

1 Introduction

The ubiquitous and pervasive computing paradigm [11] requires proper systems support for non-intrusive adaptability of context-aware applications and the platforms they run on. Self-adaptable context-aware systems promise to fill in this gap as they spontaneously modify their functionality in response to changing demands. Context-aware and self-adaptable systems reconfigure (1) the *runtime infrastructure* that manages computational resources and provides the basic functions to deploy and run applications, (2) the *context middleware* that retrieves, aggregates and utilizes context information for non-intrusive application behavior, and (3) the composition or behavior of *user applications* to accommodate to resource variability, changing environments, user preferences, etc. The number of (re)configurations in each of these layers separately can be large and result in an exponentially growing number of combinatorial variants for self-adaptivity of the whole system. The question that poses itself and is investigated in this paper is how correct and consistent self-adaptation can be achieved for the overall system when each layer is responsible for its own domain-specific self-adaptation. In this paper we study a multi-dimensional dependency and conflict resolution mechanism to find a stable and consistent reconfiguration. Projection and intersection in a multi-dimensional reconfiguration space are used to restrict the ways of how domain-specific self-adaptation can be combined. Conflict resolution helps to maintain the system integrity and usability by avoiding endlessly alternating self-adaptations and reconfigurations leading the system into an unusable state.

The remainder of this paper is organized as follows. In section 2 we describe self-adaptivity in the runtime infrastructure, the context middleware and the applications. Section 3 discusses how integrity issues may arise due to dependencies between these layers and describes the multi-dimensional dependency and conflict resolution mechanism that permits per layer self-adaptability while achieving stable reconfigurations for the overall system. We then describe related work in section 4 and conclude with section 5.

2 Self-adaptation in context-aware systems

A *context-aware system* [3] usually involves a runtime infrastructure for deploying context-aware applications enhanced with a context middleware that is capable of acquiring and aggregating context information on the current location and time, user preferences and activities, available devices, services and resources in the vicinity of the user. Self-adaptation [4] is a key feature of reflective systems [6] as they have the ability to contain a representation of itself and manipulate its state during its execution. Our previous work [8] discusses how context is used to adapt component-based services to heterogeneous devices and networks.

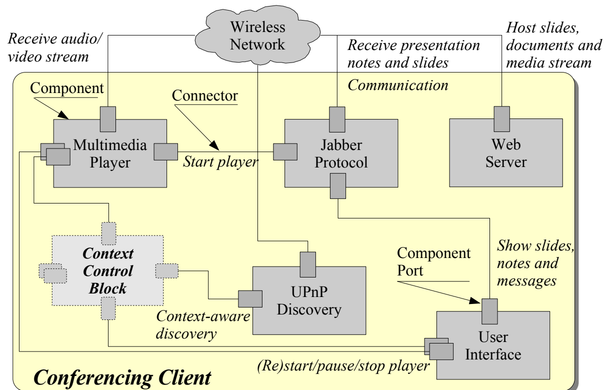

Figure 1. The conferencing client

2.1 Application self-adaptation

Applications are modeled as a composition of mandatory, optional and alternative inter-connected components [10]. Figures 1 and 2 show the composition of a Jabber-based and multimedia-enabled conferencing client that supports web based document sharing. Examples of behavioral and structural self-adaptation that the application supports include amongst other:

- At home, the media player will automatically increase the volume when playing one of your favorite songs.
- Replace/remove the *Video Decoder* component according to video input and free processing power.
- Replace the *Jabber Protocol* component with another instant messaging protocol component.

2.2 Context middleware self-adaptation

Our context middleware is also component-based and is represented as the *Context Control Block* in Figure 1. It involves the following context managing subcomponents:

- **Input:** These components gather data from information providers, such as sensors, stored user preferences and profiles, databases and other third parties.
- **Filter:** These components filter out irrelevant or old data or compare the accuracy of different input components that provide the same kind of information.
- **Persistency:** A context repository provides persistency for a concept's attributes as well as the ontologies that model the concept's relationships.
- **Transformation:** These components translate from one representation to another, e.g. temperature in ^{o}C to ^{o}F, or location in *GPS coordinates* to *city*.
- **Reasoning:** This component derives new information by combining known context concepts and derivation rules, e.g. current location based on agenda and time.
- **Dissemination:** This component provides the application with the requested context information.

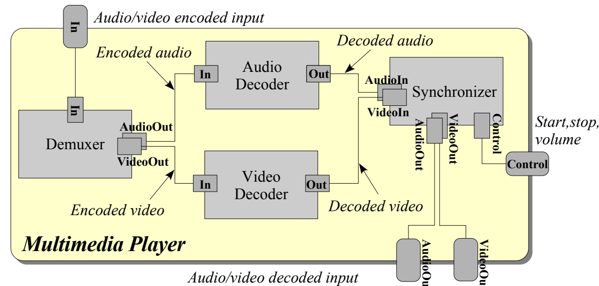

Figure 2. The multimedia player

It is self-adaptable and can whenever appropriate (un)load its own context managing subcomponents. The composition of our context middleware and its self-adaptation capabilities are discussed more in detail in [7].

2.3 Runtime infrastructure self-adaptation

The runtime infrastructure provides the basic functions to deploy and run component compositions, which include both the user applications and the context middleware. Self-adaptivity takes place at this level to provide uniform access to platform specific *I/O devices*, *hardware resources* and *resource monitors*. Self-adaptation in the runtime infrastructure mainly involves resource-awareness:

- Decrease the CPU clock frequency when no application is active or when the system is idle.
- Reduce the display backlight to extend the autonomy of the device when running on battery.
- Load support for and provide access to the capabilities of the WiFi resource.

Access to the resource monitors, I/O devices and hardware resources is enabled on demand when required by the applications or the context middleware and disabled when they are no longer needed.

3 Multi-dimensional dependency and conflict resolution

In this section we discuss how integrity issues may arise due to dependencies between the runtime infrastructure, the context middleware and the user applications and how they may result in conflicting concurrent or alternating self-adaptation. We therefore describe a multi-dimensional dependency and conflict resolution mechanism that permits per layer self-adaptability while achieving stable and consistent reconfigurations for the overall system. Projection and intersection in a multi-dimensional reconfiguration space are used to restrict the ways of how domain-specific self-adaptation can be combined. Conflict resolution helps

to maintain the system integrity and usability by avoiding endlessly alternating self-adaptations and reconfigurations leading the system into an unusable state.

3.1 Dependencies and conflicting behavioral and structural self-adaptivity

Self-adaptation within the context-aware applications can trigger self-adaptations in the context middleware and in the runtime infrastructure, and vice versa.

3.1.1 Behavioral and structural self-adaptivity dependencies between multiple layers

Assume the conferencing client enables the *Video Decoder* component and asks the context middleware to be informed of when the user is moving around in order to disable the video stream and not distract the user. The context middleware adds the location *input* component to its composition which in turn requests the runtime infrastructure to activate the WiFi *resource access* module in order to use the *Service Set IDentifier* (SSID) and the *Received Signal Strength Indication* (RSSI) of WiFi access points with known positions.

Vice versa, the runtime infrastructure may decide to scale down the CPU clock frequency when the battery is running out of power. Due to the decreased processing power the conferencing client removes the *Video Decoder* component as it cannot keep up with decoding the video stream in real time.

3.1.2 Alternating self-adaptations jeopardizing the integrity and usability of the system

It is not difficult to see how things go wrong due to hidden dependencies when combining the two scenarios from above. Alternating self-adaptation can occur as follows:

1. The context middleware informs the client application that enough free CPU power is available.
2. *Application self-adaptation*: The multimedia player adds the *Video Decoder* to its composition.
3. The battery usage (% per time unit) increases due to the high CPU load caused by the decoder.
4. *Runtime infrastructure self-adaptation*: The runtime infrastructure scales down the CPU speed.
5. *Application self-adaptation*: The multimedia player removes the *Video Decoder* as it can not decode at the required frames per second.
6. The continuously high CPU load disappears and as a result of this the battery usage goes down again.
7. *Runtime infrastructure self-adaptation*: The runtime infrastructure re-enables the CPU at full speed.

This process repeats itself endlessly and can be avoided if the *Video Decoder* component were able to state a battery usage requirement for each system.

3.1.3 Concurrent self-adaptations causing competition for shared computational resources

Assume the context middleware informs the multimedia player that enough processing power and bandwidth is available. In that case the multimedia player enables the *Video Decoder* component to play a live video stream, and the context middleware will notify the decoder when its user is moving.

The way the context middleware manages location-awareness is completely abstracted from the application. Assume the context middleware instantiates a computationally intensive *input* component that makes use of WiFi signal strength triangulation based on SSID and RSSI information and a remote database to match with well-known positions. In this case there is a competition for bandwidth and processing power between decoding the live video stream on the one hand and estimating the current position and fetching remotely available location samples on the other hand. These resource dependencies can only be determined at runtime as the positioning is completely hidden from the application. Moreover, the same multimedia player self-adaptation will not always trigger the same context middleware self-adaptation when the required context managing components are already active for another application and ready to be reused.

3.2 Modeling self-adaptation in the multi-dimensional reconfiguration space

As mentioned before, a context-aware system consists of multiple building blocks that are subject to structural or behavioral self-adaptation. The number of combinations of independent (re)configurations in each of the three layers can grow exponentially. However, self-adaptivity of the whole system is constrained by deployment dependencies on the components and modules:

- **Resource dependencies**: Each component requires computational resources to be deployed, including free memory and processing power and possibly also some bandwidth, storage, battery autonomy, etc.
- **Component dependencies**: Some components rely directly on the presence of another component to operate correctly (e.g. mandatory components in a composition). Depending on the availability of alternatives for a mandatory component, we either have a 1-to-1 dependency, or otherwise an M-to-N dependency.
- **Contextual dependencies**: Components may rely on context provided by *input* or *transformation* components from the context middleware.

Note that optional components represent themselves in the multi-dimensional reconfiguration space as isolated building blocks or as an isolated cluster of building blocks. As such, they are easy targets to free up resources.

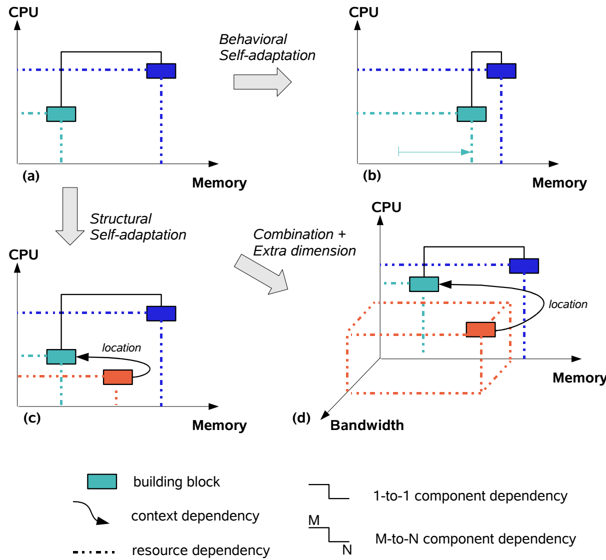

Figure 3. Modeling the multi-dimensional reconfiguration space

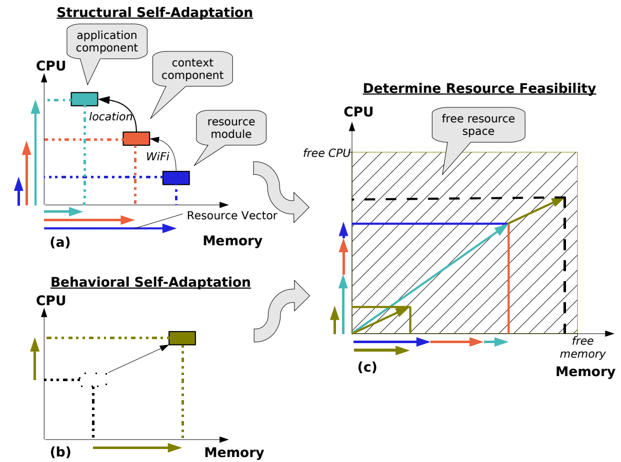

Figure 4. Resolving resource dependencies

Figure 3 shows in subfigure (a) an initial system configuration with 2 components and projection onto two resource dimensions. The connector in between the 2 components (represented with a solid line) defines a 1-to-1 component dependency. Each projection onto one of the dimensions (represented with a dotted line) specifies a resource dependency. In subfigure (b) behavioral self-adaptation is visualized as a component shifting in the multi-dimensional reconfiguration space. Subfigure (c) demonstrates structural self-adaptation due to an additional location *input* component as a contextual dependency (represented with a curved arrow). In subfigure (d) both kinds of self-adaptation are combined. The figure also demonstrates an expansion of the multi-dimensional system reconfiguration space to model the extra resource deployment dependency.

3.3 Resource feasibility analysis for concurrent structural and behavioral self-adaptation

Reconsider the self-adaptation scenarios that were mentioned in Section 2. Assume the *Multimedia Player* composite component decides to include the location-aware *Video Decoder* component into its composition. The SSID and RSSI based *input* component for location-awareness and the WiFi *resource access* module are activated to fulfill the contextual dependency of the *Video Decoder* component. This structural self-adaptation is demonstrated in Figure 4 in subfigure (a). Assume that as part of the same reconfiguration process another component is subject to behavioral self-adaptation which increases its CPU and memory usage as shown in subfigure (b). Both self-adaptations have an impact on the available resources. The *resource vectors* in subfigure (c) demonstrate the impact of both self-adaptations. The hatched area visualizes the intersection of the available processing power and memory and models the *free resource space*. It spans the same dimensions as the reconfiguration space itself. The cumulated sum of all resource vectors, defined as the *cumulated resource vector*, should point into this free resource space for self-adaptation to be feasible from a resource perspective. The dimensions of the cumulated resource vector can be negative. This means that the availability of this particular resource will increase after self-adaptation.

If the end point of the cumulated resource vector points into the free resource space it only guarantees that such a reconfiguration can exist. It does not tell how it can be achieved from the current configuration. Self-adaptation of the whole system is a sequential process of structural and behavioral self-adaptation steps at each layer. The order of this sequential process is determined by the fact that each self-adaptation step should be able to take place given the available resources. This is illustrated by the *stable* and *unstable resource points* in Figure 5. Stable resource points reside in the free resource space and thus can be achieved through stepwise self-adaptation, while unstable resource points identify a resource conflict as they reside outside the free resource space and thus can never be reached.

At runtime, each layer in the context-aware system is able to inspect the multi-dimensional reconfiguration space. If one layer, e.g. a user application, wishes to self-adapt but requests assistance from another self-adapting layer, e.g. the context middleware, it marks its new reconfiguration as ten-

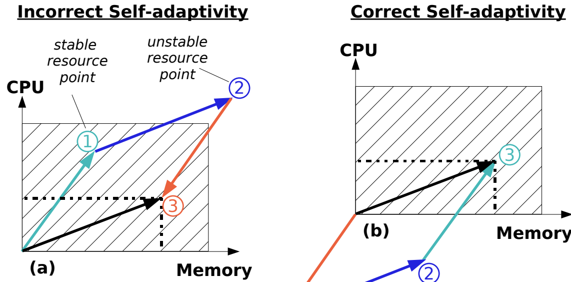

Figure 5. Resource points determining the order of self-adaptation

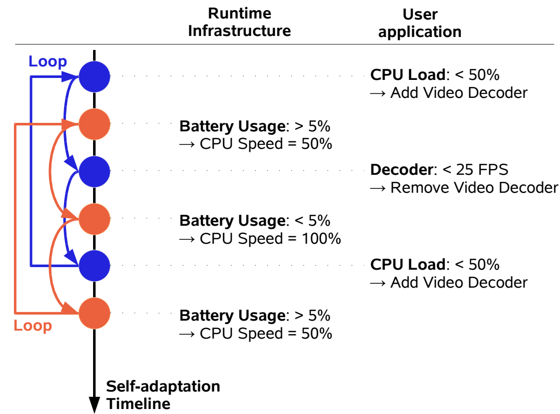

Figure 6. Alternating self-adaptations in multiple layers due to reoccurring self-adaptation conditions and actions

tative and waits for approval of the other layer. This approval ensures that its own self-adaptation does not conflict with the self-adaptation of the layer on which it depends. Later on, a correct ordering of self-adaptation steps is determined. This approach solves the problem of concurrent self-adaptations causing competition for shared computational resources.

3.4 Analyzing recurrent and alternating self-adaptations

One of the scenarios demonstrated alternating self-adaptations due to a hidden resource dependency between the battery usage and the continuously high CPU load caused by the *Video Decoder* component. Alternating self-adaptations that jeopardize the integrity of the system usually involve recurrent overlapping self-adaptations in multiple layers. Recurrent self-adaptations within one layer are caused by a bad self-adaptation strategy in the layer itself.

Multi-layer alternating self-adaptations cannot be avoided solely with the approach discussed above. However, if each layer keeps track of when and why it self-adapted, then recurrent and alternating self-adaptive behavior can be detected. Figure 6 visualizes the time dimension that illustrates the evolution of the self-adapting context-aware system. It shows the reconfiguration conditions and actions for both the runtime infrastructure and the user application. Both layers suffer from recurrent self-adaptations, which are illustrated by the two loops at the left. Alternating self-adaptations occur when the following conditions are met:

- At least two layers suffer from recurrent self-adaptations, i.e. they show reoccurring self-adaptation conditions and actions in a short time span.
- These recurring self-adaptations overlap both in the time domain as well as in the resource domain.

Loop detection is performed at the same time when resource feasibility is analyzed. It ensures that recurrent self-adaptations will not occur indefinitely. However, the last question that remains is which action from the overlapping self-adaptations will conquer: *'Add Video Decoder'* or *'CPU Speed = 50%'*? This decision is based on the priority of the self-adapting actions of each layer:

$$runtime > context > applications$$

In this case, the end result would be that the CPU speed is downscaled to 50% of its full speed and the *Video Decoder* component is removed from the multimedia player.

3.5 Evaluation

Our approach to self-adaptation for a context-aware system (runtime infrastructure, context middleware and applications) has the following benefits:

- Self-adaptation of each layer is specified and managed independently from one another. This decreases the overall adaptation complexity.
- The three layers of abstraction enable optimized deployment of the context middleware and provide runtime and context middleware agnostic self-adaptation capabilities to the applications.
- Deployment dependencies are automatically resolved. Conflicting self-adaptivity behavior is detected in advance before causing any usability issues.

Unfortunately, there are also an important drawback. Although independent self-adaptation for each layer has its advantages, it puts the burden on the developer to design a self-adaptation strategy for the application that works correctly under all circumstances. Moreover, our approach makes it more difficult to blame misbehaving applications. Unfair applications can occupy all resources, hoping that other context-aware applications will self-adapt to give up some of their resources.

4 Related work

In recent years, many researchers have addressed the issue of context-awareness and self-adaptive middleware. This research has greatly improved our knowledge of how context-aware and self-adaptive middleware can accommodate to new functionality requirements, performance optimization, variable runtime conditions and changing environments. Providing a detailed review of the state of the art on context-awareness and self-adaptability is beyond the scope of this paper. Instead, we focus on those contributions that are most related to the work presented in this paper.

Puppeteer [2] is a system for adapting component-based applications in mobile environments. Compared to our work Puppeteer also takes advantage of the component-based nature of the applications to perform adaptation. The difference with our work is that their goal is to adapt applications in order to achieve reductions in user-perceived latencies in two office applications without modifying the applications themselves.

The BBN QuO system [5] also supports applications that adapt to resource variability allowing users to define operation regions. The runtime systems monitors the application and execution environment and activates application handlers when the application changes operation region. Compared to our work, context-awareness is not being considered for application adaptation.

Sousa et al. [9] compare their work with other systems that adapt their computing strategies in reaction to bandwidth, memory, CPU and power variation. They claimed their work is an improvement as it was not restricted to adaptation of one component at a time, but they also tackled multi-component integration, configuration and reconfiguration. Our work also deals with multi-component reconfiguration, but subdivides these components into three layers of abstractions to separate self-adaption concerns into domain specific reconfigurations. Resource dependencies are resolved transparently and feasibility of system self-adaptation is automatically ensured by means of a correct order of stepwise reconfigurations.

CARISMA [1] is another context-aware reflective middleware system for mobile applications, but the crucial difference between this work including other approaches and our proposal is that our work decreases the adaptation complexity by separating the self-adaptation behavior into three independent layers of abstraction. Any dependencies that may arise during the reconfiguration of the system are resolved automatically.

5 Conclusion

In this paper we discussed structural and behavioral self-adaptation of a component-based system subdivided into three layers: the runtime infrastructure, the context middleware and the user applications. Each layer relies on its own notions of introspection and intercession to achieve domain specific reconfiguration. We identified inter-layer dependencies that may lead to concurrent self-adaptations causing competition for computational resources and alternating self-adaptations jeopardizing the usability of the system.

We proposed a multi-dimensional resource dependency and conflict resolution mechanism to achieve stable and consistent self-adaptations. Projection and intersection in a multi-dimensional reconfiguration space are used to restrict the ways of how domain-specific self-adaptation can be combined. Conflict resolution helps to maintain the system integrity by avoiding self-adaptations leading to an unusable state.

References

[1] L. Capra, W. Emmerich, and C. Mascolo. Carisma: Context-aware reflective middleware system for mobile applications. *IEEE Trans. Software Eng.*, 29(10):929–945, 2003.

[2] E. de Lara, D. S. Wallach, and W. Zwaenepoel. Puppeteer: Component-based adaptation for mobile computing. In *USITS*, pages 159–170. USENIX, 2001.

[3] A. K. Dey. Understanding and using context. *Personal Ubiquitous Comput.*, 5(1):4–7, 2001.

[4] R. Laddaga. Active software. In *IWSAS' 2000: Proceedings of the first international workshop on Self-adaptive software*, pages 11–26. Springer-Verlag New York, Inc., 2000.

[5] J. P. Loyall, R. E. Schantz, J. A. Zinky, and D. E. Bakkenl. Specifying and measuring quality of service in distributed object systems. In *The First IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 43, 1998.

[6] P. Maes. Concepts and experiments in computational reflection. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 147–155. ACM Press, 1987.

[7] D. Preuveneers and Y. Berbers. Adaptive context management using a component-based approach. In N. Alonistioti and L. Kutvonen, editors, *Proceedings of 5th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS2005)*, Lecture Notes in Computer Science (LNCS). Springer Verlag, June 2005.

[8] D. Preuveneers and Y. Berbers. Automated context-driven composition of pervasive services to alleviate non-functional concerns. *International Journal of Computing and Information Sciences*, 3(2):19–28, August 2005.

[9] J. P. Sousa, V. Poladian, D. Garlan, and B. R. Schmerl. Capitalizing on awareness of user tasks for guiding self-adaptation. In J. Castro and E. Teniente, editors, *CAiSE Workshops (2)*, pages 83–96. FEUP Edições, Porto, 2005.

[10] C. Szyperski. *Component Software: Beyond Object-Oriented Programming, 2nd edition*. Addison-Wesley and ACM Press, 2002.

[11] M. Weiser. The Computer for the Twenty-First Century. *Scientific American*, pages 99–104, Sept. 1991.