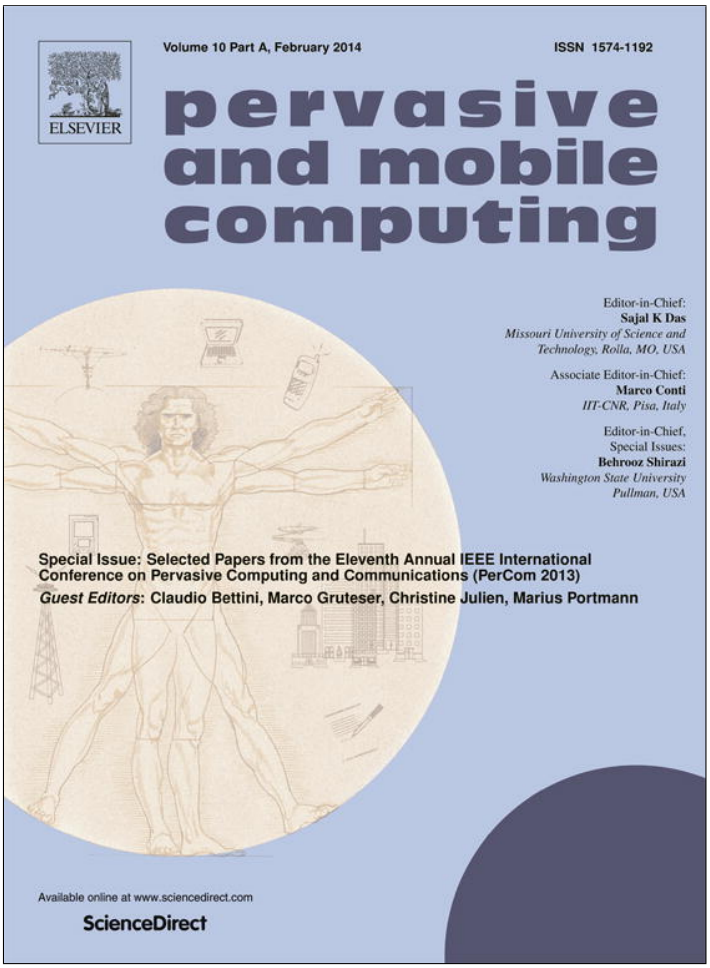


Pervasive and Mobile Computing 10 (2014) 51–65

Contents lists available at ScienceDirect

Pervasive and Mobile Computing

journal homepage: www.elsevier.com/locate/pmc

COMITY: A framework for adaptation coordination in multi-platform pervasive systems

Sebastian VanSyckel [a,*], Dominik Schäfer [a], Verena Majuntke [a], Christian Krupitzer [a], Gregor Schiele [b], Christian Becker [a]

[a] University of Mannheim, Schloss, 68161 Mannheim, Germany
[b] DERI, National University of Ireland, IDA Business Park, Galway, Ireland

ARTICLE INFO

Article history:
Available online 25 October 2013

Keywords:
Application coordination
Interference detection and resolution
Context-aware computing
Pervasive environments

ABSTRACT

Pervasive applications are designed to support users in their daily lives. In order to provide their services, these applications interact with the environment, i.e., their context. They either adapt themselves as a reaction to context changes, or adapt the context via actuators according to their needs. If multiple applications are executed in the same context, interferences are likely to occur. In this paper, we present COMITY—a framework for interference management in multi-platform pervasive systems. Based on contracts specifying an application's interaction with the context, the framework automatically detects interferences and resolves them through a coordinated application adaptation.

1. Introduction

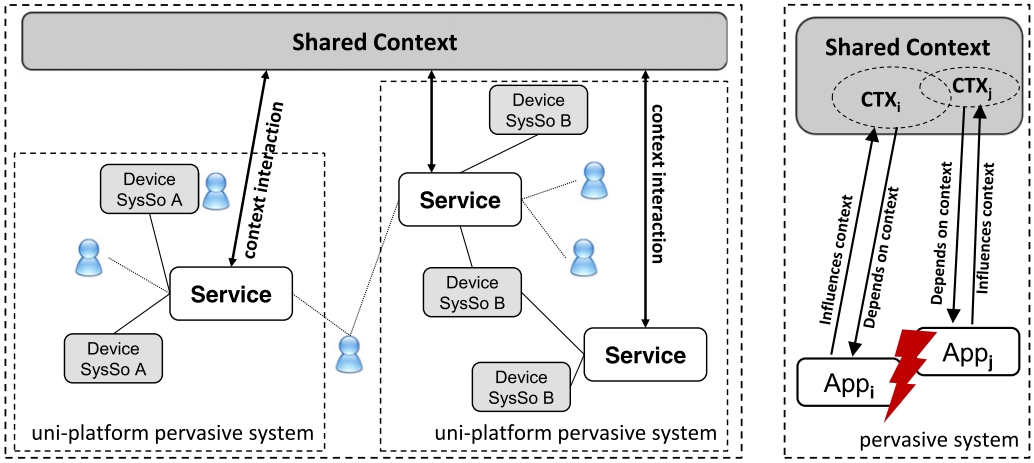
Intelligent applications in pervasive systems interact with their context, i.e., they are context-aware and context-altering. An important characteristic of context-aware applications is their ability to adapt to changes in their context. Adaptation can either be done by adapting the application's behavior, e.g., adjusting the brightness of a display or volume of an audio output, or by changing its structure, e.g., switching from audio output to a graphical user interface. As a consequence, applications that share the same context can influence each other. If applications are not aware that other applications are influenced by their context interaction – e.g., if an application dims the light, the video based input of an other application can degrade in quality – *interferences* can occur. Handling such interferences requires coordination of applications. In the past, we have presented an initial framework based on PCOM [1] in [2] which was able to detect interferences and to resolve them within the PCOM system using built-in adaptation mechanisms (cf. [3,4]).

In this paper, we present a general approach to application coordination. We extend existing systems by *context contracts* and an *adaptation interface*, enabling a thin middleware layer the coordination across different pervasive systems. The context contracts serve as a basis to detect interferences and to resolve them through a coordinated application adaptation. In [5], we presented an initial approach to interference resolution that found a solution with minimal number of adaptations, but was not applicable for systems with more than a couple of applications, due to the real-time restriction of pervasive systems. In this paper, we present algorithms for interference resolution that focus on high responsiveness, i.e., finding a solution as fast as possible, as well as algorithms that maximize the global utility of the system. The results of our evaluation show that both approaches outperform our initial approach significantly and are suitable even for larger systems.

The remainder of the paper is structured as follows. First, we describe our system model and discuss interferences in multi-platform pervasive systems in Section 2. In Section 3, we present our middleware-based approach to adaptation

* Corresponding author. Tel.: +49 6211812617.
E-mail address: sebastian.vansyckel@uni-mannheim.de (S. VanSyckel).

1574-1192/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.pmcj.2013.10.006



(a) Multi-platform pervasive system. (b) Interference.

Fig. 1. System model.

coordination. More precisely, we present our overall framework in Section 3.1, the extensions to pervasive system software that are necessary for a cross-platform coordination in Section 3.2, and the coordination process in Section 3.3. In Section 4, we describe our approach to interference resolution, before we discuss implementation details and evaluate our framework in Section 5. Finally, we close with a conclusion and an outlook on future work in Section 7.

2. Interferences in multi-platform pervasive systems

A *pervasive system* consists of a set of users and devices. These devices cooperate in order to provide services to the users. To realize a pervasive system, devices are equipped with respective system software, such as Aura [6], Gaia [7], and BASE/PCOM [8]/[1]. A *uni-platform pervasive system* is a pervasive system in which all devices are equipped with the same system software. Our work focuses on *multi-platform pervasive systems* which are illustrated in Fig. 1(a). A multi-platform pervasive system emerges if two or more uni-platform pervasive systems share the same physical space and, therefore, the same context space. To determine the physical space of a pervasive system, we assume the existence of a location model such as [9] or [10]. The location model provides a symbolic reference for physical spaces like buildings, floors, rooms, etc.

In these spaces, services are provided to the users by *pervasive applications*. A pervasive application is defined by the following three characteristics:

1. *Distribution*: A pervasive application is distributed among multiple devices. It makes use of available resources and functionalities, which form the *functional configuration* of the application.
2. *Context-interactivity:* A pervasive application interacts with its context. On the one hand, it is able to obtain context information and to translate that information into configuration decisions. On the other hand, it has the ability to influence its context, e.g., via actuators.
3. *Adaptivity:* A pervasive application has the ability to adapt itself to changes in its context. In dynamic environments, this enables applications to continue providing their service, if necessary, using a different functional configuration.

The parallel execution of such pervasive applications poses challenges in multi-platform pervasive systems. The problems arise from the fact that pervasive applications interact with a shared context. As a consequence, they are directly related with each other via their context and can have a significant impact on each other. Fig. 1(b) illustrates such a situation in which two applications interact with a shared context. Consider the situation in which application App_i has changed the context according to its needs. Afterwards, application App_j is started and discovers that the shared context does not satisfy its requirements. Consequently, App_j also adapts the context according to its needs. This action changes the basis on which App_i chose its active functional configuration. Since its current configuration is not anymore viable in the changed context, App_i is forced to react, leaving it with two options: It can adapt the context again according to its own needs or it can adapt itself. The first option may result in a cycle where the two applications take turns adapting the context, which results in an oscillation of the context between two states and with each change one of the applications is not able to operate—comparable to a deadlock. The second option may prove to be suboptimal because another configuration may not satisfy the user's requirements. Moreover, it may be possible that no viable functional configuration can be found at all for App_i.

The situation discussed above is a general problem in multi-user pervasive systems. Applications interact with the shared context. They make configuration decisions based on context states and adapt the context according to their needs, without considering that other applications may be executed in parallel. We refer to the described problem as an *interference*. An interference is an application-induced context that forces another application to react. Such interferences are not necessarily a problem, as context changes and the following adaptation are part of the natural behavior of pervasive systems. However,

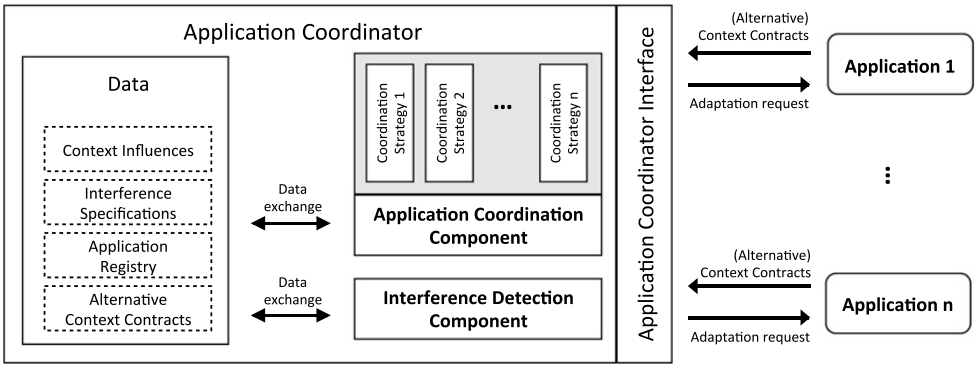


Fig. 2. Coordination framework.

there are situations – as described above – in which interferences have to be handled in order to ensure the productivity of a system. Additionally, interference coordination allows to optimize the system's performance.

For our approach of adaptation coordination, i.e., in order to handle these interferences, we assume applications to be *cooperative*. That is, applications are truthful about their dependencies and influences on the shared context, as well as compliant with the resolution process. The specifics of this cooperation are discussed in Section 3.2. Furthermore, we assume that each application is able to compute possible alternative functional configurations for a given context, as discussed in [11].

3. Middleware-based adaptation coordination

In this paper, we propose an approach for adaptation coordination in multi-platform pervasive systems. A major challenge for such a cross-system approach is the integration of applications running on various platforms. This means that platform-specific services, such as context management, cannot be used. Consequently, we assume only means of communication, such as remote procedure calls, and service discovery. Instead of platform-specific services, we derive the following two application capabilities or *system extensions*:

1. *Context contracts:* Applications must be able to specify their interaction with the context and provide their adaptation options to the coordinating entity. For this, we briefly introduced the abstract concept of context contracts in [2]. In this paper, we discuss these contracts in detail and provide a formal description on how they are modeled (cf. Section 3.2.1).
2. *Adaptation interface:* In order to resolve detected interferences, the coordinating entity must be able to instruct applications to adapt. For this, we define a simple adaptation interface in Section 3.2.2.

By implementing both the context contracts and the adaptation interface, existing systems are integrated in our approach to adaptation coordination.

3.1. Coordination framework

In order to handle interferences, we developed the coordination framework shown in Fig. 2. The basic idea of the coordination framework is the realization of a middleware layer that detects and resolves interferences between applications in different uni-platform pervasive systems.

To ensure management of application-specific interferences, each application registers at the framework and provides *context contracts*. A context contract describes the interaction with the shared context. It depends on the functional configuration of the application. The context contract consists of the application's *interference specification* and its *context influences*. The interference specification defines the context states, which lead to an interference for the application. The context influences specify how the application influences the shared context. For interference detection, each application provides an active context contract, which reflects its active functional configuration. For interference resolution, each application provides a set of alternative context contracts, which constitute the application's alternative functional configurations.

Interference detection is realized by the *interference detection component*. The process of interference detection involves the evaluation of all active interference specifications with regard to the current context. It is triggered every time the set of interference specifications or the context changes. An interference is detected if an interference specification is satisfied by the current context. In this case, a description of the interference is composed. The description includes the satisfied interference specification, the contributing context and all involved applications. Once the description is created, the interference resolution process is triggered by invoking the *application coordination component*. Interference resolution is a two-staged process. First, an *interference resolution plan* is computed according to a coordination strategy that is set for the framework. Here, we can either use the first solution found or maximize the global utility, as further described in Section 4. The resolution plan determines how applications have to adapt in order to resolve a detected interference. The plan is computed based on the active and alternative context contracts the applications have registered. We describe the details

of our interference resolution approach in Section 4. Once a resolution plan has been obtained, the framework instructs applications to adapt according to the plan.

3.2. System extensions

Our approach to adaptation coordination supports multi-platform pervasive systems. Hence, we can only use common features of system software, such as communication and service discovery. In order to still achieve this support, we need to introduce two system extensions, which we describe next.

3.2.1. Context contracts

A *context contract* defines the interaction of an application with its context depending on a functional configuration. That is, a context contract is the specification of all context relations – dependencies and effects – of a functional configuration. For application coordination, each application provides the *active context contract* for its current configuration, and at least one *alternative context contract* for alternative functional configurations. The active context contract is required for interference detection, whereas the list of alternative context contracts is used for interference resolution.

A challenge in specifying these context contracts is to ensure that the shared context is addressed by all applications in a common way. Further, specifying the contracts requires a priori knowledge from the developers regarding the specific environment, e.g., available context services, as well as the effects of their application on certain contexts, and vice versa. This can be achieved by using ontologies, such as [12] or [13], and suitable environment modeling and simulation tools. We built a prototype simulation environment for our evaluation, in which we can selectively activate contracts – and thereby change the context – to study the behavior of the system. On this note, sophisticated IDE plug-ins for the entire development and debugging process of creating interactive pervasive environments are highly desirable and, for COMITY, will be addressed in future work. However, both ontologies and modeling/ simulation tools for developers are not in the scope of this paper.

Context contracts consist of two mandatory parts, the interference specification and the context influences. Both are discussed in detail in the following. Additionally, each contract can have a utility value in order to model its value regarding, for example, its benefit for the user.

Interference specification. The first part of a context contract is the *interference specification*. It defines the context states that pose an interference for an application. Interference specifications have a major influence on the complexity and efficiency of the application coordination process. To select a formal model for the specifications, three aspects need to be considered: The *expressiveness* of the logic, the *efficiency of statement evaluation*, and the test on *unsatisfiability of the set of interference specifications*. We chose to base our model on monadic predicate calculus [14], as it proves to have a good balance between all aspects. With respect to expressiveness, the logic extends propositional logic by quantifiers and unary predicates. This allows to address elements in sets of objects and to make statements about them. In contrast to first-order predicate logic, which allows predicates of arbitrary arity, monadic predicate logic only supports the use of predicates of arity one. This restriction to monadic predicates prevents the modeling of relationships. However, the restriction has a positive impact on the efficiency of statement evaluation. With monadic predicate logic, the complexity for evaluating a statement is $O(|\varphi||\mathcal{A}|)$ – in contrast to $O(|\varphi||\mathcal{A}|^{|\varphi|})$ for first-order predicate logic – where $|\varphi|$ is the number or attributes in the formula φ expressing the statement, and $|\mathcal{A}|$ is the number of possible assignments for each of these attributes defined by the structure $A = (U_A, I_A)$, where U_A is the universe of A and I_A is an interpretation function that assigns (1) each n-ary predicate symbol P an n-ary predicate symbol over U_A, (2) each n-ary function symbol f an n-ary function symbol over U_A, and (3) each variable x an element from the universe U_A [15]. In other words, $|\varphi|$ is the number of relevant contexts and $|\mathcal{A}|$ is the accumulated number of valid states for those contexts. Further, in contrast to more expressive models, such as first-order predicate logic, a check for unsatisfiability is decidable. This check is used to ensure that solutions to interferences can be found in general. A more detailed discussion on this can be found in [15].

Fig. 3 shows an example interference specification of a presentation application. The interference specification models three context states that the application encounters as an interference. The first one is a temperature that is below 19 °C. The second specifies an interference, if a presentation takes place in the environment and the lights are not dimmed. The third one models an interference when a presentation is held and the audio volume is greater than 45 decibels.

The interference specifications need to be defined by the application developer. One possibility is to automatically infer the interference specification from an application's context-action rules, i.e., the rules that describe how an application has to act in a certain context. Furthermore, user requirements towards the context can be modeled as interferences. If a user feels disturbed by loud noises while talking on the phone, the respective context state can be added to the interference specification.

Context influences. The second part of the context contract are the application's *context influences*. Context influences explicitly specify the effects an application has on the shared context. They are determined through the resources and actuators that the application uses. In active context contracts, the context influences specify the actual effects on the shared context. For alternative context contracts, the actual context influences may not be known before its instantiation. An alternative context contract can state that it will affect the context with audio, but it will probably not know with which intensity. Hence, applications can specify expected context influences which may cover a range of values. Once the alternative contract becomes active, the explicit context influences are set.

```
BEGIN
        temperature < 19.0 Celsius
    OR
        activity = video_presentation  AND  light.intensity != dimmed
    OR
        activity = video_presentation  AND  audio.volume > 55 decibels
END
```

Fig. 3. An example interference specification.

```
CI  =  {activity = video_presentation,
        light.intensity = dimmed,
        audio.type = speech,
        audio.volume = 55 decibels}
```

Fig. 4. An example context influence definition.

```
interface Instructable {

    void adaptToCC(ContextContract cc);

}
```

Fig. 5. The adaptation interface.

Fig. 4 shows an example context influence definition of a video presentation. The context influences state that the application sets the activity of the environment to video presentation, dims the lights, and outputs speech with an intensity of 55 decibels.

3.2.2. Adaptation interface

The second system extension is the *adaptation interface*. As mentioned before, we assume applications to be cooperative. One aspect of that is that they implement the *Instructable* interface shown in Fig. 5.

The interface defines a single functionality. It enables the coordination framework to instruct an application to switch into an alternative functional configuration. As a parameter, it passes the respective context contract, which has been determined to be part of the interference resolution plan (see Section 4), to the application. The application then instantiates this contract, respectively any functional configuration that complies with this context contract. As the applications report every switch between functional configurations to the coordinator, no return value is necessary.

3.3. Coordination process

Fig. 6 gives an overview of the application coordination process. First, each application registers for coordination at the central coordinator instance and, in the same step, specifies its current, i.e., active context contract (1). The coordinator adds this information to its respective data structures (2) and replies with the application's specific ID (3). Concurrently, the coordinator triggers the interference detection (3). In fact, it does so after each change to the set of active contracts, e.g., when a new application registers or an existing application changes its functional configuration. In case of an interference, the coordinator starts the resolution process (4). This subprocess computes an interference resolution plan. In case one exists, the coordinator sends adaptation instructions to the applications (5). Otherwise, the coordinator currently advises the application causing the interference, i.e., the application that last entered the system or changed its functional configuration, to pause its execution, at which point a manual adaptation by the users is necessary. Anytime after (3), applications can add alternative context contracts (6) or activate one of their previously added context contracts (7) using their ID. Adding alternative context contracts does not trigger the interference detection, as they have no effect on the shared context. However, activating a contract triggers the detection, as it may follow an adaptation that has different context influences then the previously active contract.

To illustrate the process further, consider a presentation application as the lone application in the environment ($app1$) with an active context contract $cc11$ consisting of the interference specification and context influence shown in Figs. 3 and 4. Further, the same application has an alternative context contract $cc12$, in which the context influence specifies an audio volume of 50 decibels. Subsequently, an e-book reader application enters the environment and registers with the coordinator, i.e., it registers and informs the coordinator of its active context contract, and receives its ID $app2$. Its active context contract $cc21$ specifies an interference for the context audio volume at an intensity greater than 50 decibels, i.e., $audio.volume > 50\ decibels$. (For the simplicity of this example, we assume that the context influences of $cc21$ do not interfere with the interference specifications of either $cc11$ or $cc12$.) With this change to its set of active contracts,

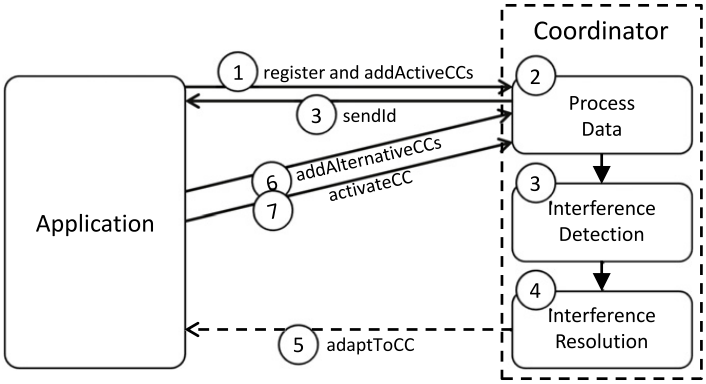


Fig. 6. Application coordination process.

the coordinator triggers the interference detection process and discovers the interference regarding the context audio volume. Hence, the resolution process is started. The solution in this simple example is obviously an adaptation of the presentation application (*app*1) to a functional configuration that fulfils its alternative contract *cc*12, and no adaptation of the e-book reader application (*app*2). Consequently, the coordinator instructs the presentation application (*app*1) to adapt to its alternative contract *cc*12 using the *Instructable* interface shown in Fig. 5. After the application adapts accordingly, it activates the contract *cc*12 at the coordinator.

Next, we discuss the problem of interference resolution in more detail and present our approach.

4. Interference resolution

The process of resolving an interference can be split into two steps: Computing an interference resolution plan, and instructing applications to adapt according to that plan. The latter is achieved via a call to the adaptation interface (see Section 3.2.2). The former is a complex task and requires a systematic approach. In this section and for the purpose of describing our approach, we assume that an interference resolution plan exists for any given interference.

Given an interference and the set of all active and alternative context contracts, an *interference resolution plan* is a list that assigns a context contract to each active application in the pervasive system. If each application fulfils its respective assignment, the interference is resolved and an interference-free system state emerges. In case an application already fulfils the assigned contract, no change is required of it. Otherwise, it must adapt accordingly.

In order to determine such a plan, the application coordination component searches for a context contract for each application such that the detected interference is resolved and no new interferences are created. That means, an assignment needs to be found for each application such that (i) the context influences of the application do not satisfy any then active interference specification – which can change as part of the resolution plan – and (ii) the new interference specification of the application is not satisfied by the then existing context.

In the following, we analyze the problem of finding an interference plan in detail and formalize the problem by modeling it as a *constraint satisfaction problem*. Subsequently, we present various resolution algorithms for finding a solution as fast as possible, or maximizing the global utility, respectively.

4.1. Interference resolution as CSP

Computing an interference resolution plan is a complex task. The complexity stems from the fact that context influences and interference specifications are strongly related to each other. On the one hand, the context influences change the context and, therefore, the basis on which interference specifications are evaluated. On the other hand, the interference specifications restrict the possible context influences of every other application. Changing the context contract of an application changes its context influences as well as its interference specification. Thus, using an alternative context contract may show that the contract's interference specification is satisfied by the current context, as well as the contract's context influences satisfy an existing interference specification.

In order to reason about the complexity of computing an interference resolution plan, we modeled the problem as a constraint satisfaction problem (CSP) [16] which is defined as follows:

Constraint Satisfaction Problem (CSP) A constraint satisfaction problem is a triple (V, D, C) where $V = \{V_1, \ldots, V_n\}$ is a finite set of variables and $D = \{D(V_1), \ldots, D(V_n)\}$ is a set of finite domains such that $D(V_i)$ is the finite set of potential values for V_i. Furthermore, $C = \{C_1, \ldots, C_k\}$ is a finite set of constraints where each C_l is a pair (t_l, R_l) with $t_l = (v_{l_1}, \ldots, v_{l_m})$ being an m-tuple of variables and R_l being an m-ary relation over D. A solution of an instance of a CSP is a function $f : V \to D$ such that $\forall(t_l, R_l)$ with $t_l = (v_{l_1}, \ldots, v_{l_m})(f(v_{l_1}), \ldots, f(v_{l_m})) \in R_l$.

Based on the previous definition, the problem of computing an interference resolution plan can be modeled as a constraint satisfaction problem as follows:

Let V be the set of applications $App = \{App_1, \ldots, App_n\}$ which are active in the environment, and let D be the set of finite domains $CC(App) = \{CC(App_1), \ldots, CC(App_n)\}$, where $CC(App_i) = \{(CI_{i_1}, IS_{i_1}), \ldots, (CI_{i_m}, IS_{i_m})\}$ is the finite domain of App_i, namely the finite set of possible context contracts (CC) for App_i where CI are the context influences and IS is the interference specification of the contract. Furthermore, let $C = (t, R)$ be the single constraint with $t = (App_1, \ldots, App_n)$ and $R = \bigcup_{i=1}^{n} CI_{i_j} \cup CTX_{nat}(\bigcup_{i=1}^{n} IS_{i_j}) \models 0$. Thus, a solution to the problem of computing an interference resolution plan is a selection of a context contract for each application, such that the union of the context influences of all applications in combination with the natural context (CTX_{nat}) does not satisfy the union of all interference specifications. Please note that we have explicitly not addressed the possibility of automatically pausing one or more applications in order to create an interference-free state in case no resolution plan could be determined. In that event, we currently defer to manual adaptation.

4.2. Resolution algorithms

In our preceding work [5], we introduced a variable approach to interference resolution that found a resolution plan with the minimal number of necessary adaptations. To do so, we started from the last functional set of context contracts and appended the ones causing the interference at the end. The algorithm alternated the context contracts of those application first, which caused the interference. From that point, we used a *backtracking*-based approach combined with a pruning technique similar to *backjumping* in order to find a resolution plan. Even though a resolution plan with minimal adaptations is desirable, the approach is not applicable for systems with more than a couple of applications in it. This stems from the fact that the approach is not able to gather and use information about the relation between the domains of the different applications and, hence, is not able to prune enough of the search space.

In this paper, we present algorithms for finding an interference resolution plan that are closer to the traditional approaches for solving CSPs and make use of information gained in the process. During calculation, we need to test the *consistency* – consistency in terms of a CSP equals an interference-free state in our system – of various combinations of context contracts. For this, we activate these context contracts at the coordinator, adjust the recorded context data according to the context influences in those contracts, and trigger the interference detection process. However, the actual context is not affected and the active functional configurations of the applications do not change until a plan is found and the applications are instructed to adapt.

In our new approach, we implemented a set of tree-based algorithms to solve the CSP. Each of these algorithms manages a set of context contracts for applications $App_0, \ldots, App_i$ that is gradually extended with contracts for applications $App_{i+1}, \ldots, App_n$ until a *consistent* combination of contracts is found. At the beginning, this set only contains the contract of the first application. After each extension, the combination is checked for interferences and, if none are found, it is extended again. Thus, we have a so called partial solution at each step and a full solution after extending the combination of contracts for each application.

On the one hand, this approach increases the number of interference detection steps. After each change to the combination of contracts, the new combination must be checked for consistency. On the other hand, the approach minimizes the overhead in our previous approach that arises from checking several combinations that, for example, only differ towards App_n, even though there are interferences caused by applications near App_0. In other words, the new approach adds contracts only to consistent subsolutions, whereas the previous approach may try to solve the CSP by only varying contracts of some applications, even though the contracts of the other applications alone would not be consistent either.

Two important aspects in pervasive systems are responsiveness and service quality. Consequently, it is important to resolve an interference as fast as possible in order to ensure the functionality of the system. However, the global utility of a solution should also be as high as possible to enhance the usability of an application. Achieving both at the same time is not easy, as the latter requires comparing multiple solutions, if more than one exists. Hence, we implemented two different sets of algorithms that each focus on one of the two aspects.

First, we implemented a set of traditional approaches for solving CSPs in order to find a resolution plan as fast as possible. They disregard the utility of contracts and terminate after finding the first consistent state. Based on these, we then implemented a set of *branch and bound* algorithms in order to optimize the global utility. Instead of terminating after the first solution, they store the utility of the consistent state and resume, while pruning parts of the search tree that cannot yield better solutions. Next, we describe both sets of algorithms in more detail.

4.2.1. Algorithms for high responsiveness

For finding a consistent state as fast as possible, we implemented a set of algorithms described in [17], namely *backtracking* (BT), *conflict-directed backjumping* (CBJ), *explicit forward checking* (FC), and the hybrid approach *explicit forward checking with conflict-directed backjumping* (FC–CBJ). All use variations of two basic functions, *label* and *unlabel*. The label function is a forward step that tries to extend the existing consistent partial solution with a contract from the domain of the next application. For this, we differentiate between the *domain* and the *current domain* of an application. The domain is the entire set of contracts for that application and remains unchanged, whereas the current domain is a subset that only contains contracts that are consistent with the consistent partial solution. The unlabel function is a backward step. Backward steps are executed if there is no further extension possible for a partial solution. Hence, changes in the preceding part of the

combination of contracts are retracted until the next consistent partial solution is found. After a successful label step on the last application, the algorithm found an interference-free solution.

Backtracking (BT). Standard BT – which we implemented for baseline measurements – operates only on information about the interference state of the current combination. During the label step for application App_i, BT iterates through all contracts of App_i's current domain $CC_{current}(App_i)$ and checks the consistency of the new setting. As soon as BT finds a consistent setting, it proceeds with labeling App_{i+1}. In case a setting is not consistent, BT removes the non-fitting contract from $CC_{current}(App_i)$. If no consistent setting was found for applications $App_0, \ldots, App_i$, BT unlabels App_i. In this case, the partial solution for $App_0, \ldots, App_{i-1}$ was consistent, but could not be extended by any contract in $CC(App_i)$. Hence, the unlabel function of BT restores all context contracts of App_i, i.e., sets $CC_{current}(App_i) := CC(App_i)$, removes the active contract of App_{i-1}, and proceeds with labeling App_{i-1}. BT terminates with a solution if it finds a consistent setting for $App_0, \ldots, App_n$, or without a solution if $CC_{current}(App_0)$ is empty.

While extending the partial solutions, it is possible to gather information about the relation between the contracts of the various domains. Based on this information, different forward and backward stepping strategies are possible in order to decrease the number of steps necessary to find a solution. Of these strategies, we implemented the most *informed backtracker* CBJ, the *forward move* strategy FC, as well as their hybrid FC–CBJ. The entire set of strategies can be found in [17].

Conflict-Directed Back Jumping (CBJ). CBJ uses the same labeling approach as BT, i.e., it checks the consistency up to the current application App_i. However, it acquires information along the way about the consistency between the contracts in $CC_{(App_i)}$ and those part of the partial solution consisting of $App_0, \ldots, App_{i-1}$. This information is stored for each application in so-called *conflict sets*. In case no consistent setting could be found for the partial solution and App_i, CBJ unlabels applications $App_i, \ldots, App_h$ where $h < i$ and h is the deepest variable in the conflict set of App_i. Hence, CBJ jumps over all the combinations possible from $App_{h+1}, \ldots, App_{i-1}$ that BT checks but cannot lead to a solution, as the inconsistency is caused by App_h and App_i. Further, CBJ carries the conflict set of App_i upwards to App_h during unlabeling. This way, CBJ is able to jump backwards multiple times, if necessary. In contrast, *backjumping* is only able to jump back once and then defers to standard backtracking. During unlabeling, CBJ restores the conflict sets and current domains of the applications it jumped over. Again, the algorithm terminates with a solution if it finds a consistent setting for $App_0, \ldots, App_n$, or without a solution if $CC_{current}(App_0)$ is empty.

Explicit Forward Checking (FC). FC uses a special labeling algorithm in order to decrease the number of steps necessary to find a solution. The unlabel function, on the other hand, is equal to the one of BT. While iterating through the contracts of $CC_{current}(App_i)$ during the label step for App_i, it already checks the consistency between the current contract (CI_{i_k}, IS_{i_k}) and all contracts of the succeeding applications $App_{i+1}, \ldots, App_n$. In case such a consistency check fails, for example between (CI_{i_k}, IS_{i_k}) of App_i and (CI_{j_l}, IS_{j_l}) of App_j, FC removes contract (CI_{j_l}, IS_{j_l}) from $CC_{current}(App_j)$. As a result, the current domains of the future applications become smaller and FC has to check less contracts moving forward. If the current domain for any application in $App_{i+1}, \ldots, App_n$ results in being empty, no solution is possible that includes contract (CI_{i_k}, IS_{i_k}). In this case, FC reverts all changes to $CC_{current}(App_{i+1}), \ldots, CC_{current}(App_n)$ that were caused by labeling App_i with (CI_{i_k}, IS_{i_k}), and (CI_{i_k}, IS_{i_k}) is removed from $CC_{current}(App_i)$. As soon as FC finds a contract in $CC_{current}(App_i)$ that is consistent with at least one contract from each future domain $CC_{current}(App_{i+1}), \ldots, CC_{current}(App_n)$, it proceeds with labeling App_{i+1}. In case $CC_{current}(App_i)$ becomes empty during labeling, FC unlabels App_i. FC also terminates as soon as it finds a solution or $CC_{current}(App_0)$ becomes empty.

Explicit Forward Checking with Conflict-Directed Back Jumping (FC–CBJ). FC–CBJ combines the forward move approach of FC with the informed backtracking of CBJ. It iterates through smaller current domains during labeling and jumps over more applications while unlabeling. Hence, we expect FC–CBJ to have the best performance in our evaluation.

4.2.2. Algorithms for maximizing utility

The overall goal of pervasive computing is supporting the users in fulfilling their tasks. That means ideally, we would always like to maximize the utility of the system. This is especially the case if an adaptation is necessary anyways. In order to achieve such an optimization in our system, we first added a utility value to each context contract. These might, for example, reflect the preferences of the users towards the different functional configurations of an application. Subsequently, the CSP becomes a constraint optimization problem and the goal is to find a solution with maximum utility. Currently, the utility values of each contract are a fixed value between 0.1 and 1.0. In future work, however, we plan on making them dynamic in order to reflect utilities that depend on the context influences of other active applications.

For finding the interference resolution plan that leads to the maximum global utility, we modified the algorithms described above following the *branch and bound* approach. That is, rather than terminating after finding a solution, the modified algorithms continue their search until all solutions were found or all potential solutions were discarded based on their utility. Further, they follow a breadth-first search (BFS) approach. During labeling, the algorithms iterate through the current domain based on the utility of the contracts in descending order. Finally, the algorithms keep track of the global utility of each (partial) solution – which is the sum of the utility of all active context contracts – and backtracks as soon as the known maximum cannot be outdone. That is, after successfully labeling App_i and calculating the utility of the partial solution including applications $App_0, \ldots, App_i$, the algorithms estimate the final global utility using the closest lower bound $|App_{i+1}, \ldots, App_n| * 1.0$. If the estimate is smaller than the best known value, the algorithm discards that partial solution and backtracks. As an example, assume we have labeled the first three of five applications with utilities of 0.6, 0.7, and 0.5, respectively, accumulating to a global utility of thus partial solution of 1.8. As there are two unlabeled applications

Algorithm 1 O-FC-CBJ Resolution

```
1:  procedure O-FC-CBJ(n, status)
2:      consistent ← true
3:      status ← "unknown"
4:      maxUtility ← 0
5:      solution ← null
6:      btFlag ← false
7:      i ← 1
8:      sortCurrentDomainsByUtilityDSC()
9:      while status = "unknown" do
10:         if consistent then
11:             i ← o-fc-cbj-label (i, consistent, btFlag)
12:         else
13:             i ← o-fc-cbj-unlabel (i, consistent, btFlag)
14:         end if
15:         if i > n then
16:             if currentUtility > maxUtility then
17:                 maxUtility ← currentUtility
18:                 solution ← getHeadsOfCurrentDomains()
19:             end if
20:             btFlag ← true
21:             i ← i - 1
22:             deactivateCC(currentDomains[i].head)
23:             currentDomains[i].removeHead
24:             consistent ← currentDomains[i] ≠ null
25:         else if i = 0 then
26:             if solution ≠ null then
27:                 status ← "optimal solution"
28:             else
29:                 status ← "impossible"
30:             end if
31:         end if
32:     end while
33:     return solution
34: end procedure
```

left, and the maximal possible utility of each of them is 1.0, we add 2.0 to the global utility of the partial solution, and our closest lower bound heuristic is 3.8. If the global utility of the best solution up to now is lower than 3.8, we continue with labeling the fourth application. If it is higher or equal, we will not be able to surpass it by labeling the remaining two applications and can therefore discard the current partial solution. We modified all four algorithms BT, CBJ, FC, and FC–CBJ to become O–BT, O–CBJ, O–FC, and O–FC–CBJ. In the following, we describe these modifications by means of the algorithm O–FC–CBJ. The others follow the same structure. In the description of the algorithms, we use the procedures *checkForward* and *undoReductions*. The first prunes the domains of the yet unlabeled applications, and the second undoes those changes in case of an unlabel. For further details regarding these procedures and how the conflict set is managed, please see the description of FC and CBJ in the preceding section, or [17].

Algorithm 1 formulates the resolution procedure of O–FC–CBJ. In it, we added three variables, namely *maxUtility* that stores the maximal known utility, *solution* that stores the solution with the maximal known utility, as well as *btFlag*. We need this flag in order to indicate, whether the *unlabel* procedure should follow its respective backtracking approach, e.g., conflict-directed backjumping, or simply backtrack one level. We describe the details of this flag momentarily. Further, we realize the BFS approach, i.e., choosing the domain with the highest utility first, by sorting the current domain list in descending order and always retrieving the head element. This way, we have $n * O(m \log m)$, $m = \max(|CC_{current}(App_i)|)$ once instead of $O(|CC_{current}(App_i)|)$ at each labeling. However, this is only possible due to the static utility values. With dynamic values, a traditional BFS approach at each labeling is required. Finally, the algorithm does not terminate as soon as a solution has been found, i.e., $consistent = true$ and $i > n$. Instead, it updates *maxUtility* and manipulates itself to continue its search. That is, we deactivate the current contract of the last application at the coordinator, remove it from the current domain list, and decrement the level i. Subsequently, the algorithm proceeds in one of two scenarios: (i) The current domain of the last application is not empty. In this case, it proceeds as if it had lust successfully labeled the next to last application, by labeling the last application again. (ii) The current domain of the last application is empty. Here, the *btFlag* comes into play. In this case, an unlabeling is necessary. However, we do not want to unlabel due to a conflict, i.e., the respective backtracking strategy does not apply. Instead, we want to find the next node in the search tree. Hence, we proceed with standard backtracking, until we have successfully labeled an application again, which is indicated by the *btFlag* (see Algorithm 2).

Algorithm 2 shows the O–FC–CBJ label function. While the current domain is not empty and we have not found a consistent state yet, we apply the algorithm-specific labeling steps. In this instance, we check forward and manage the respective conflict sets. In case we find a consistent extension, we estimate its maximal utility and only proceed if that estimation is greater than the known maximal utility. If the estimation – which is always an overestimate – is less than the maximal known utility, but we have a consistent extension, we need to backtrack one level instead of following the

Algorithm 2 O-FC-CBJ Label

```
1:  procedure O-FC-CBJ-LABEL(i, consistent, btFlag)
2:      consistent ← false
3:      while currentDomains[i] ≠ ∅∧ not consistent do
4:          consistent ← true
5:          activateCC(currentDomains[i].head)
6:          for j = i + 1 → n do
7:              consistent ← checkForward(i,j)
8:          end for
9:          if not consistent then
10:             deactivateCC(currentDomains[i].head)
11:             currentDomains[i].removeHead
12:             undoReductions(i)
13:             conf-set[i] ← union(conf-set[i], past-fc[j-1])
14:         end if
15:     end while
16:     if consistent then
17:         btFlag ← false
18:         if currentUtility + estimate ≥ maxUtility then
19:             return i+1
20:         else
21:             consistent ← false
22:             return i
23:         end if
24:     end if
25:     return i
26: end procedure
```

Algorithm 3 O-FC-CBJ Unlabel

```
1:  procedure O-FC-CBJ-UNLABEL(i, consistent, btFlag)
2:      if btFlag then
3:          h ← i-1
4:      else
5:          h ← max(max-list(conf-set[i],max-list(past-fc[i]))
6:      end if
7:      for j = h + 1 → i do
8:          deactivateCC(currentDomains[j].head)
9:          conf-set[j] ← 0
10:         undoReductions(j)
11:         restoreCurrentDomains(j)
12:     end for
13:     undoReductions(h)
14:     currentDomains[h].removeHead
15:     consistent ← currentDomains[h] ≠ null
16:     return h
17: end procedure
```

respective backtracking approach. That is, in case of O–FC–CBJ, for example, we unlabel App_{i-1} instead of the maximum level in the conflict set of App_i (see Algorithm 3). Again, we do this by maintaining *btFlag* that indicates, whether the next unlabel is due to a non-consistent setting or a discard based on the utility estimation of that setting. Hence, in case of a successful labeling, we set *btFlag* to *false*.

In Algorithm 3, we either conduct the respective backtracking strategy, or a standard backtracking step, depending on *btFlag*. That is, level *h* is either the maximum level from the conflict set of App_i, or simply one level above *i*. In the latter case, the CBJ-specific *for*-loop is not executed and *o-fc-cbj-unlabel* behaves like *o-fc-unlabel* would. Ignoring the *undoReductions(i)* call, it behaves like *o-bt-unlabel*.

After presenting our two types of interference resolution algorithms, we evaluate their performance in the following section.

5. Implementation and evaluation

In this section, we describe the implementation of our framework COMITY and evaluate the performance of our approach to interference resolution. For a detailed evaluation of the other components of the framework, such as memory requirements of COMITY and performance of the interference detection, please see [5].

5.1. Implementation

In order to evaluate our approach in a reasonable scenario, we set up a pervasive system using our system software BASE [8]. BASE is a middleware that has been designed for pervasive systems. It has a lightweight but extensible core,

which enables its operation on resource-poor devices, such as embedded systems, but also supports costly functionalities running on full-fledged devices, such as desktop computers. Devices which are equipped with BASE are able to detect each other and form a spontaneous network. In order to build and execute pervasive applications, BASE models functionalities and device capabilities as services and provides a uniform access. Each (remote) service can be accessed via local proxies implementing a defined interface. Moreover, BASE enables remote communication while shielding applications from the underlying communication technologies, interoperability protocols and communication models.

We implemented the coordination framework COMITY as a BASE service making coordination accessible to all applications in the system. While doing this, we only used BASE services that are found in most middleware-based systems, such as service discovery and remote procedure calls. Consequently, COMITY is not limited to our BASE/PCOM platform.

5.2. Evaluation

In [5], we found that the interference resolution process is by far the biggest factor in the runtime of the overall coordination process. As mentioned earlier, our previous approach to interference resolution was a backtracking-based algorithm with an additional pruning mechanism that operated on the last functional combination of configurations and found a solution with the minimal number of adaptations. However, we found that approach only to be applicable either for pervasive systems with few applications, or interferences with reduced complexity, respectively. Hence, we developed a new backtracking approach to interference resolution with the goal of improving runtime. Further, we presented a branch and bound-based modification to the new approach in order to find the adaptation resolution plan that maximizes the global utility of the system. We evaluated these new resolution approaches on three identical desktop PCs each with an Intel Core 2 Quad Processor Q6600 CPU (2.40 GHz per core) and 4 GB RAM running a 64-bit Ubuntu 12.04 operating system.

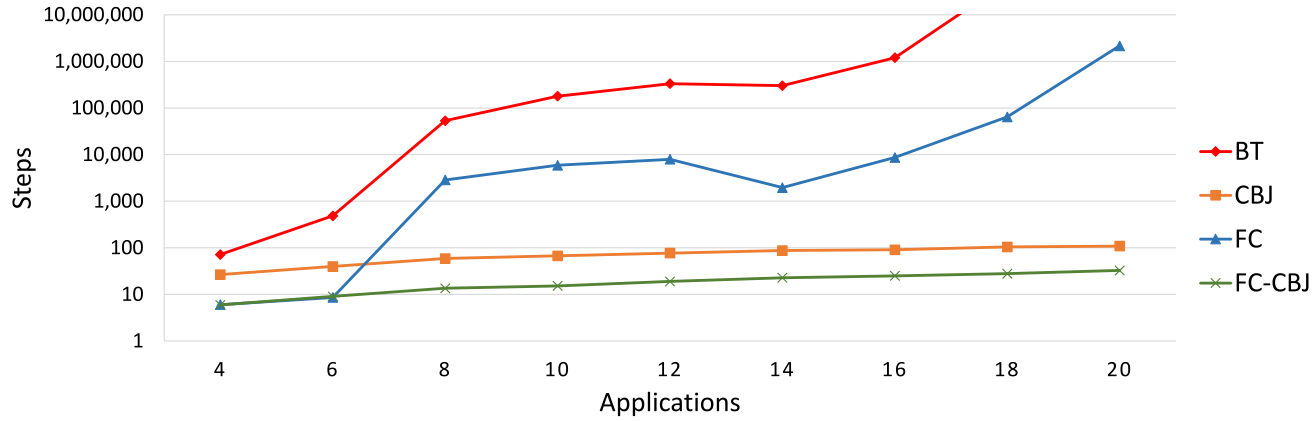
In total, we have eight interference resolution procedures (BT, CBJ, FC, FC–CBJ, O–BT, O–CBJ, O–FC, and O–FC–CBJ). We evaluate all eight using 50 randomly generated test cases with the following parameters defining the pervasive system as well as the interference resolution problem: (i) the number of applications $n = \{4, 6, 8, 10, 12, 14, 16, 18, 20\}$, and (ii) the number of context contracts per application that can resolve the interference $r = \{m/2, m/4\}$, with the number of context contracts per application fixed at $m = 8$. For each application, the context contracts that can resolve the interferences are distributed randomly among the entire set of context contracts. Additionally, each contract has a random utility value out of [0.1, 1.0] in steps 0.1. Further, we fixed the number of applications that minimally need to be adapted to $a = n/2$, and the number of attributes per context contract to $|CI/IS| = 5$. Initially, two of the applications are involved in an interference. In this evaluation, the solution space is not defined by any factors. Instead, a set of applications not involved in the initial interference have to be adapted as well. That is, a combination of contracts that is interference-free for the applications initially involved, may result in interferences with applications previously not involved. As a result, the search space becomes unpredictable. For a more detailed discussion of the characteristics of the resolution problem, we again kindly refer to [5].

Fig. 7 shows the average number of steps, i.e., the number of consistency checks that are executed, required by each variation of the two sets of resolution algorithms with respect to the number of applications n and the size of the solution space defined by $r = m/2$. We use the number of steps as the first unit of measurement to show the relation between the different algorithms, regardless of the system they are running on. We will discuss runtime in ms subsequently. As expected, FC–CBJ and O–FC–CBJ, respectively, outperform the other algorithms every time. For 20 applications with 8 context contracts each – i.e., a total of $8^{20} = 1.15$ quintillion possible combinations – FC–CBJ performs, on average, 32.66 consistency checks with $r = m/2$ and 33.86 with $r = m/4$, respectively, in order to find an interference free combination. When searching for the maximum global utility, O–FC–CBJ performs, on average, 250.3 and 265.7 consistency checks, respectively.

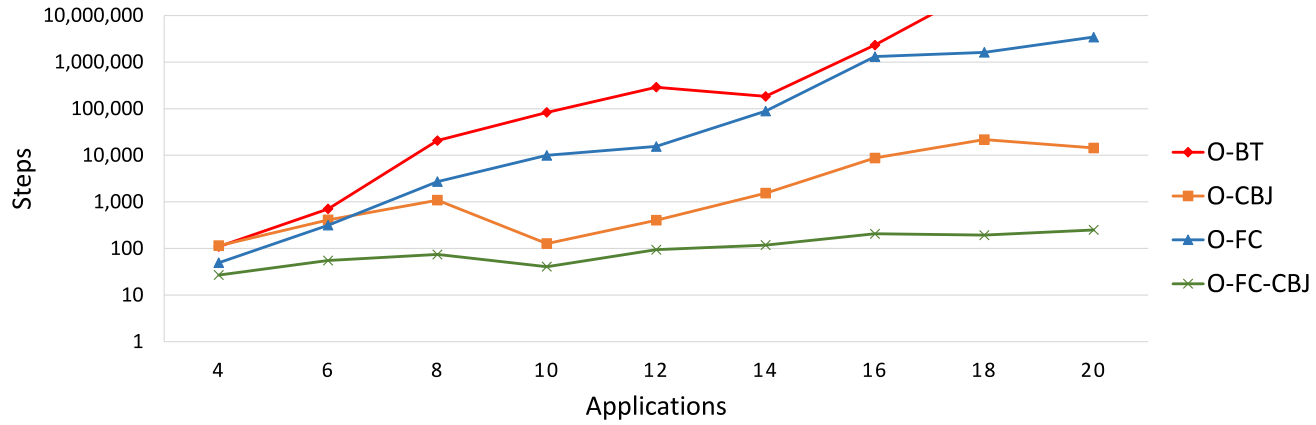
Fig. 7(a) shows the set of algorithms for high responsiveness. All algorithms show a runtime exponential in n, with some noise due to randomness. However, the growth rate for CBJ and FC–CBJ is substantially lower as for BT and FC. Additionally, we can identify strong ties between the performances of the algorithms with the same backtracking strategy. That is, BT and FC, which both backtrack only one level in case of an inconsistent subsolution, share growth characteristics, as do CBJ and FC–CBJ, which both do a conflict-directed backjump in case of an inconsistency. Hence, we conclude that the level of *informed backtracking* has the most significant effect on the performance of the algorithms. On average, the algorithms combined find interference resolution plans with between 1.16 and 1.2 adaptation more than minimally required.

Fig. 7(b) shows the set of algorithms for maximizing the global utility. Here, the performance characteristics of these algorithms is similar to those terminating after they find the first solution. One interesting finding is that with O–CBJ, the effect of the pruning heuristic greatly improves with $n \geq 10$. With less applications, the heuristic – even though the smallest possible overestimate – does not prune much of the relative search space, which shows in the number of consistency checks. Further, comparing O–CBJ with O–FC–CBJ, we see that the explicit forward checking strategy has a positive effect while searching for the maximum global utility.

Fig. 8 shows the average runtime in ms required by each variation of the two sets of resolution algorithms with respect to the number of applications n and the size of the solution space defined by $r = m/2$. Using this metric, the growth characteristics are, as one would expect, the same as with average number of steps. However, CBJ actually outperforms the other algorithms when it comes to finding one solution as fast as possible. This is due to the significant forward checking overhead of FC–CBJ. For 20 applications, CBJ takes 0.17 ms with $r = m/2$ and 0.21 ms with $r = m/4$, respectively, in order to find an interference free combination. When searching for the solution with the maximum global utility, O–CBJ initially also proves to be the fastest approach. However, O–FC–CBJ eventually outperforms O–CBJ with $n \geq 14$ and shows a significantly



(a) Algorithms for high responsiveness.



(b) Algorithms for maximizing utility.

Fig. 7. Performance of the interference resolution in *#steps*.

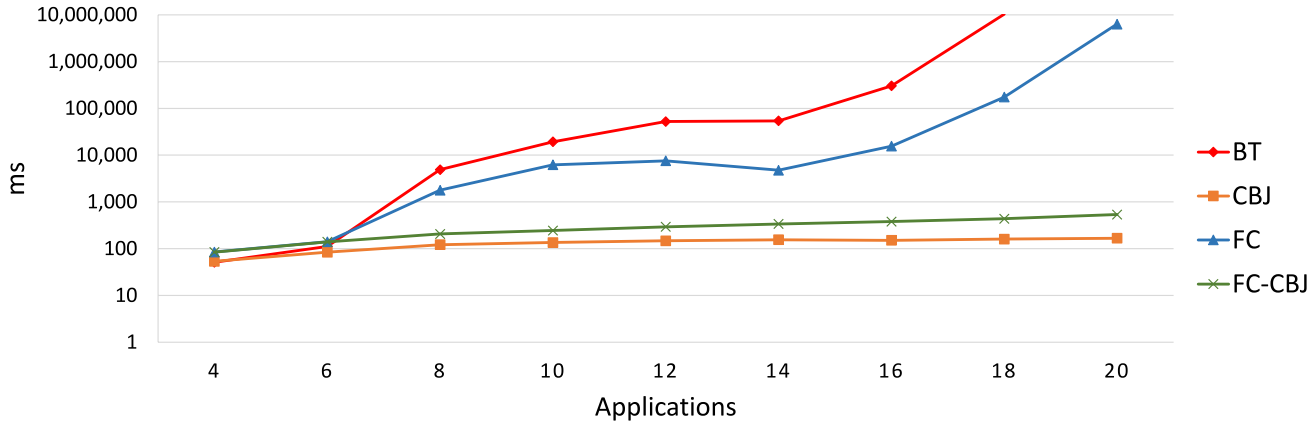
smaller growth rate. Hence, after this point, the positive effect of the forward checking strategy outweighs its overhead. In our evaluation setup and for 20 applications, O–FC–CBJ takes, on average, 2.1 s to find the optimal solution. Further, it finds the optimal solution in less than one second for as many as 14 applications.

Finally, Fig. 9 shows the runtime (in *#steps* and ms, respectively) of the best performing FC–CBJ in situations where there are no solutions for 4, 8, 12, 16, and 20 applications. For up to 20 applications in the environment, the interference resolution process determines such an unresolvable situation consistently in less than 1 s. In these situations, a manual adaptation or the termination of an application, respectively, is necessary. Currently in our prototype system, the most recent context change causing the interference is revoked. However, other strategies such as, again, optimizing the global utility or policy-based rules are also conceivable.

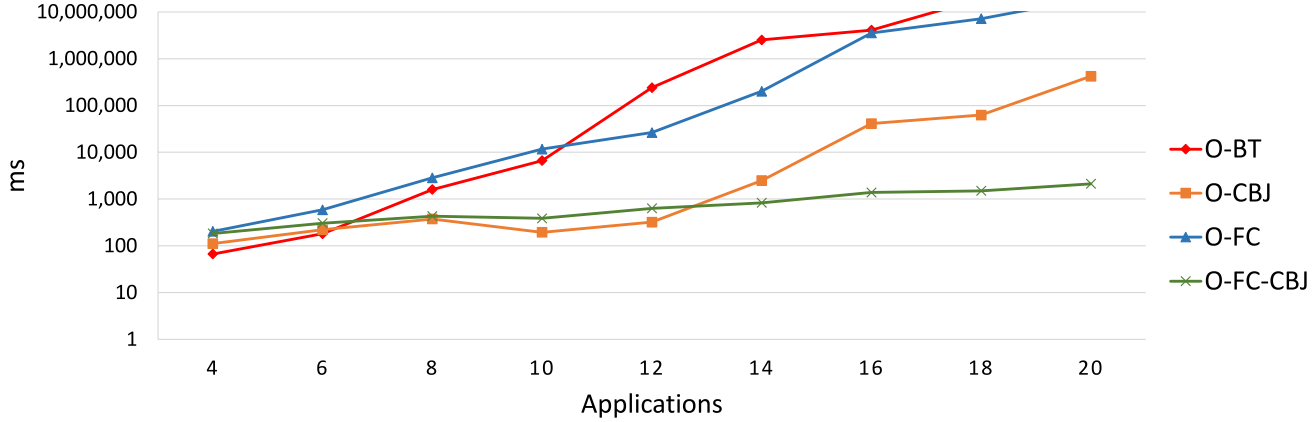
Overall, the results of our new approach to interference resolution show a significant improvement in runtime compared to our approach in [5], including the algorithms maximizing the global utility. The improvement is actually so significant, that both sets of algorithms are suitable – i.e., resolving an interference in less than one second – for real-time pervasive systems containing up to 14 applications, regardless of the complexity of the interference. Using the approach for high responsiveness, systems with 20, and presumably more, applications can also be coordinated. In comparison, our previous approach, which always found a minimal resolution plan, needed 1.77 s for eight applications, and was not suitable for systems with more than eight applications. With the new approach, however, we have to adapt approximately one application more than minimally necessary.

6. Related work

The analysis of existing work shows that subclasses of the problem of interferences have been identified and addressed under the terms interference, service interaction and conflict. Frameworks for the management of these subclasses which come close to the definition of interferences in this paper have been presented by Bortenschlager et al. [18] and Morla et al. [19]. [18] introduces the UbiCoMo infrastructure for agent coordination in pervasive systems and discusses a number of patterns to handle situations which require coordination. [19] present a general framework that allows application developers to reason about interferences offline and provides solutions to solve them. While both approaches address the problem in a general way, the considerations remain on a theoretical level and do not provide algorithmic solutions. Both



(a) Algorithms for high responsiveness.



(b) Algorithms for maximizing utility.

Fig. 8. Performance of the interference resolution in ms.

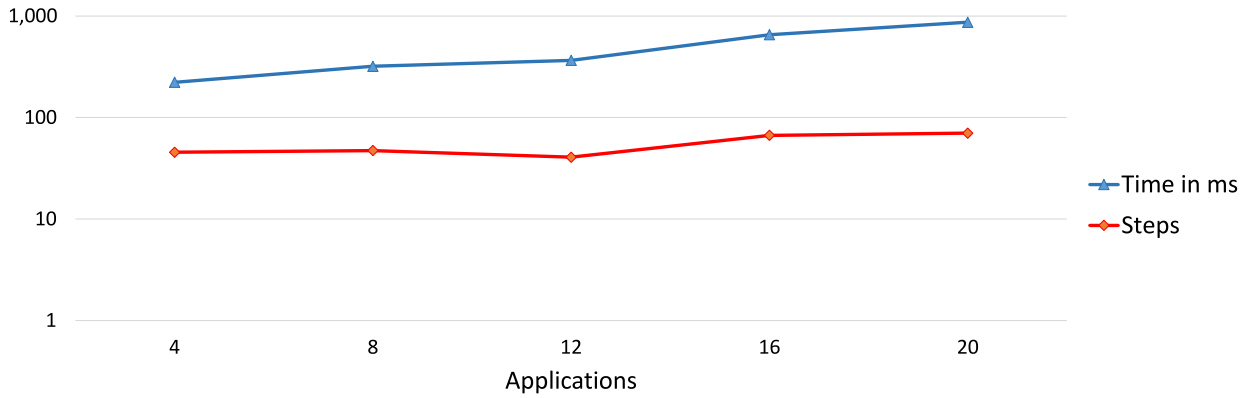


Fig. 9. Runtime of the interference resolution (FC–CBJ) without solutions.

provide a foundation for application coordination in pervasive systems by presenting a suitable infrastructure and abstract coordination patterns. However, coordination at runtime and its associated challenges are not addressed.

A variety of research work exists which explicitly focuses on the problem of interference (conflict, service interaction) resolution [20–28]. The use of priorities, for example, to solve a detected conflict (interference) has widely been investigated [20–23,29–32]. Haya et al. [21], for example, approach the resolution of concurrent request to exclusive resources by employing preemptive priority queues. A centralized mechanism is used to store action requests on resources in queues. Each action request has a predefined priority. If several requests for a resource exist in a queue, the request with the highest priority is chosen. Priority-based resolution strategies are also employed for interferences (conflicts) that occur between different users.

Shin et al. [29] dynamically assign priorities to users based on their context conflict history. If a user's context is likely to lead to a conflict according to the history, the user is assigned a low priority. Further approaches resolve conflicts based on

user preferences, e.g., [27,28,33]. These approaches are based on the idea that users have preferences towards services and how they are composed, e.g., use of specific resources. An interference (conflict) occurs if a service is accessed by multiple users or when multiple services share limited resources. The resolution of a detected conflict is achieved by computing service compositions trying to optimize user satisfaction based on preferences.

Further approaches propose a resolution process and combine several strategies which may also require the interaction with the user. For example, Shin et al. [24–26] developed a process consisting of three resolution strategies and which are applied depending on the characteristics of a detected conflict. The first two strategies support an automatic conflict resolution employing user preferences or user priorities. The third strategy is referred to as technology augmented social mediation. It presents a list of service recommendations to the users, takes their preference statements and computes a decision based on the input.

The work discussed above focuses on specific strategies for interference resolution instead of the general management of interferences. In these approaches, a resolution is either forced by the application with the highest priority, determined by a *first-come, first-served* policy, calculated based on user preferences, or a combination thereof. As a side note, such priority, policy, and preference-based strategies are very valid for many scenarios and can be integrated as coordination strategies into our framework, in similar fashion to how we maximize global utility. In the cases of priority and policy-based resolution, no calculations are necessary for resolution. The system determines which application may adapt the context in its favor, disregarding the requirements of the other applications. Even though addressing conflict resolution, such *winner-takes-all* approaches are not comparable to our work in terms of runtime, etc. In the case of user preferences-based resolution, the system calculates the adaptation automatically – ranging from simply the mean value to more sophisticated weight function-based calculations, depending on the specific system – or mediates the resolution process. The runtime of these resolution processes are mostly negligible small in comparison to our work. However, they also have some drawbacks. First, the calculated adaptation is not guaranteed to satisfy the requirements of each application. Instead, the systems calculate a *most acceptable* adaptation that satisfies the majority of users. Our resolution process, on the other hand, focuses on maximizing the number of functional applications. Second, and more significantly, the approaches above resolve conflicts per context resource, e.g., the temperature in the room or the TV channel shown. As a result, it is possible that the constraints of an application that is not functional due to the state of a resource A, still influences the state of a resource B. Using our context contracts, we specify bundles of applications constraints that are necessary for an application to be functional. If one of the constraint of a contract is not satisfiable, the others are disregarded. This differentiation is the source of the higher complexity in the calculation of a resolution in our approach.

An approach that does consider the dependencies between an applications constraints regarding multiple context resources is the graph-based conflict detection and resolution approach by Masoumzadeh et al. [32]. This approach uses a *conflict graph* to represent the current interference situation of a system and applies a resolution algorithm that uses sequences of resolution policies to find a solution. Unfortunately, the authors do not measure the actual runtime of their detection and resolution algorithms, but instead give their complexity in big O notation. However, they indicate that the resolution process is a very time consuming and should be done at compile time, if possible.

Finally, all of the discussed approaches do not address multi-platform systems. Interferences are assumed to occur and to be detected between pervasive applications which are run in a single system.

7. Conclusion and future work

In this paper we presented an approach to realize application coordination in multi-platform pervasive systems. In order for applications to be integrated, existing systems are required to implement context contracts and an adaptation interface. Based on the context contracts, interferences can automatically be detected and resolved by planning and initiating a coordinated application adaptation. For adaptation planning, we presented two types of algorithms that improved our previous results. First, we implemented a set of backtracking-based algorithms that aim at high responsiveness. Secondly, we modified those algorithms following the branch and bound concept in order to find the resolution plan that leads to the highest global utility. Our evaluation shows that our new approach to interference resolution is suitable for pervasive systems with at least 20 applications, regardless of the complexity of the interference. Furthermore, our new approach is able to find a resolution plan that maximizes the global utility in under one second for systems with up to 14 applications. This is a significant improvement over our previous approach.

The parameter that can be adapted and also has a major impact on interference resolution is the number of context contracts per application. Thus, for future work, we plan to achieve a minimal mapping of functional configurations to context contracts. Further, we want to address the development and debugging process by creating suitable tools for designing environments and applications, as well as testing the interaction of applications in a simulation testbeds. Finally, we plan on enhancing our system to support proactive adaptation, i.e., develop adaptation coordination for pervasive systems in which the applications adapt or plan adaptations based on context prediction.

Acknowledgment

This work was supported by the German Research Foundation (DFG).

References

[1] C. Becker, M. Handte, G. Schiele, K. Rothermel, PCOM—a component system for pervasive computing, in: Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications, 2004, PerCom 2004, IEEE, pp. 67–76.
[2] V. Majuntke, G. Schiele, K. Spohrer, M. Handte, C. Becker, A coordination framework for pervasive applications in multi-user environments, in: 2010 Sixth International Conference on Intelligent Environments, IE, IEEE, pp. 178–184.
[3] M. Handte, C. Becker, K. Rothermel, Peer-based automatic configuration of pervasive applications, International Journal of Pervasive Computing and Communications 1 (2005) 251–264.
[4] M. Handte, G. Schiele, V. Matjuntke, C. Becker, P.J. Marrón, 3PC: system support for adaptive peer-to-peer pervasive computing, ACM Transactions on Autonomous and Adaptive Systems (TAAS) 7 (2012) 10.
[5] V. Majuntke, S. VanSyckel, D. Schäfer, C. Krupitzer, G. Schiele, C. Becker, COMITY: coordinated application adaptation in multi-platform pervasive systems, in: Proceedings of the IEEE International Conference on Pervasive Computing and Communications, 2013, PerCom 2013, IEEE.
[6] D. Garlan, D.P. Siewiorek, A. Smailagic, P. Steenkiste, Project aura: toward distraction-free pervasive computing, IEEE Pervasive Computing 1 (2002) 22–31.
[7] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R.H. Campbell, K. Nahrstedt, A middleware infrastructure for active spaces, IEEE Pervasive Computing 1 (2002) 74–83.
[8] C. Becker, G. Schiele, H. Gubbels, K. Rothermel, Base—a micro-broker-based middleware for pervasive computing, in: Proceedings of the First IEEE International Conference on Pervasive Computing and Communications, 2003, PerCom 2003, IEEE, pp. 443–451.
[9] I. Satoh, A location model for pervasive computing environments, in: Third IEEE International Conference on Pervasive Computing and Communications, 2005, PerCom 2005, IEEE, pp. 215–224.
[10] M. Bauer, C. Becker, K. Rothermel, Location models from the perspective of context-aware applications and mobile ad hoc networks, Personal and Ubiquitous Computing 6 (2002) 322–328.
[11] K. Geihs, R. Reichle, M. Wagner, M.U. Khan, Modeling of context-aware self-adaptive applications in ubiquitous and service-oriented environments, in: Software Engineering for Self-Adaptive Systems, Springer, 2009, pp. 146–163.
[12] H. Chen, F. Perich, T. Finin, A. Joshi, SOUPA: standard ontology for ubiquitous and pervasive applications, in: The First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services, 2004, MOBIQUITOUS 2004, IEEE, pp. 258–267.
[13] P. Korpipaa, J. Mantyjarvi, J. Kela, H. Keranen, E.-J. Malm, Managing context information in mobile devices, IEEE Pervasive Computing 2 (2003) 42–51.
[14] L. Löwenheim, On possibilities in the calculus of relatives (1915), in: J. Van Heijenoort (Ed.), From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931, Harvard University Press, 1967, pp. 232–251.
[15] V. Majuntke, Application coordination in pervasive systems, Ph.D. Thesis, University of Mannheim, 2012.
[16] S.J. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, Vol. 2, Prentice Hall, Englewood Cliffs, 2010.
[17] P. Prosser, Hybrid algorithms for the constraint satisfaction problem, Computational Intelligence 9 (1993) 268–299.
[18] M. Bortenschlager, G. Castelli, A. Rosi, F. Zambonelli, A context-sensitive infrastructure for coordinating agents in ubiquitous environments, Multiagent and Grid Systems 5 (2009) 1–18.
[19] R. Morla, N. Davies, A framework for describing interference in ubiquitous computing environments, in: Fourth Annual IEEE International Conference on Pervasive Computing and Communications Workshops, 2006, PerCom Workshops 2006, IEEE, pp. 632–635.
[20] H. Lee, J. Park, P. Park, M. Jung, D. Shin, Dynamic conflict detection and resolution in a human-centered ubiquitous environment, in: Universal Access in Human–Computer Interaction. Ambient Interaction, Springer, 2007, pp. 132–140.
[21] P.A. Haya, G. Montoro, A. Esquivel, M. García-Herranz, X. Alamán, A mechanism for solving conflicts in ambient intelligent environments, Journal of Universal Computer Science 12 (2006) 284–296.
[22] A. Ranganathan, R.H. Campbell, An infrastructure for context-awareness based on first order logic, Personal and Ubiquitous Computing 7 (2003) 353–364.
[23] M. Kolberg, E.H. Magill, M. Wilson, Compatibility issues between services supporting networked appliances, IEEE Communications Magazine 41 (2003) 136–147.
[24] C. Shin, A.K. Dey, W. Woo, Mixed-initiative conflict resolution for context-aware applications, in: Proceedings of the 10th International Conference on Ubiquitous Computing, ACM, pp. 262–271.
[25] C. Shin, W. Woo, Service conflict management framework for multi-user inhabited smart home, Journal of Universal Computer Science 15 (2009) 2330–2352.
[26] F. Otto, C. Shin, W. Woo, A. Schmidt, A user survey on: how to deal with conflicts resulting from individual input devices in context-aware environments, in: Advances in Pervasive Computing 2006, Adjunct Proceedings of Pervasive 2006, 2006, pp. 65–68.
[27] I. Park, D. Lee, S.J. Hyun, A dynamic context-conflict management scheme for group-aware ubiquitous computing environments, in: 29th Annual International Computer Software and Applications Conference, 2005, Vol. 1, COMPSAC 2005, IEEE, pp. 359–364.
[28] G. Thyagaraju, S. Joshi, U.P. Kulkarni, S. NarasimhaMurthy, A. Yardi, Conflict resolution in multiuser context-aware environments, 2008 International Conference on Computational Intelligence for Modelling Control & Automation, IEEE, pp. 332–338.
[29] C. Shin, Y. Oh, W. Woo, History-based conflict management for multi-users and multi-services, in: Context2005 Workshop (Proc. of the Workshop on Context Modeling and Decision Support).
[30] M. Wilson, M. Kolberg, E. Magill, Considering side effects in service interactions in home automation-an online approach, 2008.
[31] E. Syukur, S.W. Loke, P. Stanski, Methods for policy conflict detection and resolution in pervasive computing environments, in: Proc. of Policy Management for Web Workshop, Chiba, Japan.
[32] A. Masoumzadeh, M. Amini, R. Jalili, Conflict detection and resolution in context-aware authorization, in: 21st International Conference on Advanced Information Networking and Applications Workshops, 2007, Vol. 1, AINAW'07, IEEE, pp. 505–511.
[33] C. Shin, D. Han, W. Woo, Conflict management for media services by exploiting service profile and user preference, in: Proc. ubiPCMM, 2005, pp. 48–57.