Automatic Reactive Adaptation of Pervasive Applications

Marcus Handte*, Klaus Herrmann*, Gregor Schiele+, Christian Becker+, Kurt Rothermel*
*Universität Stuttgart, +Universität Mannheim, Germany
*{firstname.lastname}@ipvs.uni-stuttgart.de, +{firstname.lastname}@uni-mannheim.de

***Abstract*-Pervasive Computing envisions seamless and distraction-free support for everyday tasks through distributed applications that leverage the resources of the users' environment. Due to the mobility of users and devices, applications need to adapt continuously to their changing execution environment. Therefore, developers need a suitable framework in order to efficiently create adaptive applications. In this paper, we present and evaluate our approach to adapting a pervasive computing application to changes during its execution. This work is based on the minimal component system PCOM [2] and on an algorithm to fully automate the initial configuration of a component-based application [11] which we have presented in earlier work. The contribution of this paper is threefold. First, we describe a number of modifications to the component model that are required to enable fully automatic adaptation. Secondly, we propose a simple yet powerful cost model to capture the complexity of specific adaptations. Thirdly, we describe an online optimization heuristic that extends our distributed configuration algorithm in order to choose to a low-cost configuration whenever the current configuration of a pervasive application requires adaptation.**

I. INTRODUCTION

Pervasive Computing aims at the realization of a paradox, namely computer systems that are both, ubiquitous and invisible for the users that interact with them. To this end, Pervasive Computing envisions seamless and distraction-free task support by combining the specific functionalities of a multitude of computer systems that are invisibly integrated into everyday objects. Thus, pervasive applications are inherently distributed, and since many everyday objects are portable, the execution environments of pervasive applications are heterogeneous and dynamic. As a result, pervasive applications must adapt continuously to their ever-changing execution environment. The need for continuous adaptation combined with the overall goal of providing applications in a distraction-free way make it practically impossible to impose the responsibility for adapting applications on the user. Instead, adaptation must be handled automatically.

To ease the development and to enable the adaptation of pervasive applications, we have created the PCOM component system [2]. With PCOM, developers specify the dependencies between their software components using contracts. At runtime, the system resolves dependencies using appropriate components available in the environment. The key idea is to define and enforce strong guarantees with respect to the set of components that constitutes an application. More specifically, PCOM aims at maintaining the invariant that all dependencies are resolved by adequate components. Providing strong guarantees despite the ever-changing execution environment can greatly simplify application development since a developer does not have to deal programmatically with unresolved dependencies.

As a first step towards enforcing this invariant, we have developed a configuration process for PCOM applications [11]. This process fully automates the initial configuration of a component-based application. In this paper, we extend this work by introducing an adaptation process that fully automates the task of adapting an application in cases where adaptation is unavoidable to maintain the invariant. To minimize the distraction resulting from adaptation, the process supports parameterization (equipping existing components with new contractual parameters) and structural adaptation (replacing existing components with new ones). The contribution of this paper is threefold. First, we describe a number of modifications to the original component model that are required to enable fully automatic runtime adaptation. Secondly, we introduce a cost model for runtime adaptation of PCOM applications. Thirdly, we present an online optimization heuristic that extends our distributed configuration algorithm.

The structure of the remainder of this paper is as follows. In the next section, we provide a brief description of the relevant concepts of PCOM. Section III presents the overall adaptation process and the modifications of the component model. Section IV presents the cost model and the optimization heuristic as an extension to our existing configuration algorithm. Section V provides an evaluation regarding the resulting overhead and the benefits. Finally, Section VI describes related approaches and Section VII concludes the paper with a summary and an outlook.

II. PCOM

Before we describe the details of the adaptation process in the next section, we briefly outline the relevant concepts of PCOM. More details can be found in [2]. PCOM components are atomic with respect to distribution. Components reside within a component container that is running on every device. Each component explicitly specifies its dependencies in a contract. This contract defines the functionality offered by the component, i.e. its offer, and its requirements with respect to local resources and other components (see Figure 1). Offered

and required functionalities are described indirectly through interface names. To model the non-functional properties, the syntactical description can be enriched with properties, i.e. typed name-value pairs for offers and typed name-value-comparator triples for requirements. Using this description, the system can automatically determine whether an offer can satisfy a requirement. An offer satisfies a certain requirement if the offer specifies a superset of the interfaces and properties contained in the requirement specification and all comparators of the requirement specification evaluate to true when the corresponding properties of the requirement and offer are compared.

A component can only be used if its local resource and component requirements can be satisfied by existing offers. Utilizing a component can lead to new requirements recursively, e.g. in Figure 1, the Converter requires two additional components. Thus, the application model supported by PCOM is a tree that starts from a root component, the so-called application anchor. Components can be embedded in multiple applications simultaneously. To enable this, components are instantiated for each usage. As soon as they are no longer used, they are automatically removed. To do this, PCOM defines a basic lifecycle that consists of a START and a STOP state. The lifecycle of the application anchor defines the overall lifecycle of the application. If the anchor is instantiated and started by the user, the system automatically resolves its contractually specified resource and component requirements recursively. If the anchor (i.e., the whole application) is stopped, the containers release all component instances and resources that have been allocated.

To resolve component requirements, each container is equipped with a remote query interface that lets another container search for matching offers. Dependencies on local resources are automatically resolved by the container that hosts the component. The resources available on a container can be strictly limited, e.g. to model exclusive resources such as input devices. Thus, using a certain component on a device might prohibit the use of another component on this device due to a lack of resources. The container guarantees that the available resources are not allocated in a conflicting way at any point in time.

The configuration process that is cooperatively performed by the containers available in a given environment ensures that only valid configurations are started. A valid configuration is defined as a tree of components starting from an anchor where all contractually specified component and resource requirements are met. Since the set of available resources and reachable devices might fluctuate at runtime, e.g. because a user unplugged an input device or because he carried a Laptop into another room, the container cannot guarantee that the configuration stays valid at all times. Therefore, the container additionally provides monitoring and signaling mechanisms as well as set of generic mechanisms that can be used to adapt an application.

The monitoring and signaling mechanisms can detect contractual changes, i.e. changes to the properties of a contract, and compositional changes, i.e. component instances that are no longer available. Contractual changes are used to express that a previously agreed set of properties can no longer be guaranteed, e.g. due to changed network properties. Compositional changes are a result of device mobility or failures.

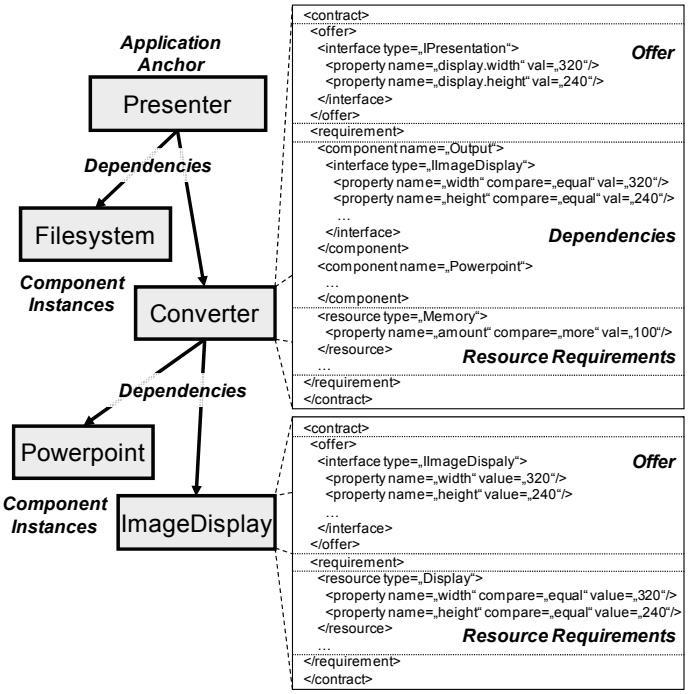

Fig. 1. PCOM Concepts

To react to these changes, the original version of PCOM provides two mechanisms. First, a component can decide to adapt its own contract in response to a contractual change in its parent component in the tree. Thus, contractual changes may propagate recursively within a sub-tree. Secondly, a component can decide to replace a component with another one. In order to support the consistent and automatic replacement, PCOM always replaces the complete sub-tree that starts from this component. Using the mechanisms proposed in [12] such a replacement works even in cases where components carry application-specific state that can no longer be accessed.

III. Adaptation Process

In the original version of PCOM, selecting a parameterization or requesting a structural adaptation is a task that must be performed programmatically within each component [2]. In response to change, a component instance can stop its execution, adapt its own contract, or request the replacement of a component it currently uses. While this approach allows fast local adaptations, adaptation on a per-component basis incurs the drawback of being inherently sub-optimal in cases where local adaptations are conflicting with other parts of the configuration. To solve this problem an adaptation must consider the effects on the complete application configuration, i.e., it must be executed globally on a per-application basis.

To adapt an application configuration globally, we rely on the following adaptation process. The high-level view of this process is straight-forward and can be explained best by walking through the lifecycle of an application as depicted in Figure 2: As soon as an application anchor is started, the configuration process described in [11] takes care of computing and starting a valid application configuration. As long as the application is executed, the component containers cooperatively monitor the application. If the configuration becomes invalid at any point in time, e.g. due to a device failure or a change in resource availability, the component system must detect this and it must initiate a global adaptation. To adapt the application, the system must compute a new valid configuration. If the current environment supports different configurations, the computation should select one that minimizes the resulting distraction. To do this, we utilize the cost model and heuristics detailed in the next section. After the new valid configuration has been computed, the existing configuration must be adapted accordingly. As soon as the new configuration is started, the system continues with monitoring until a new adaptation is necessary or the application is stopped.

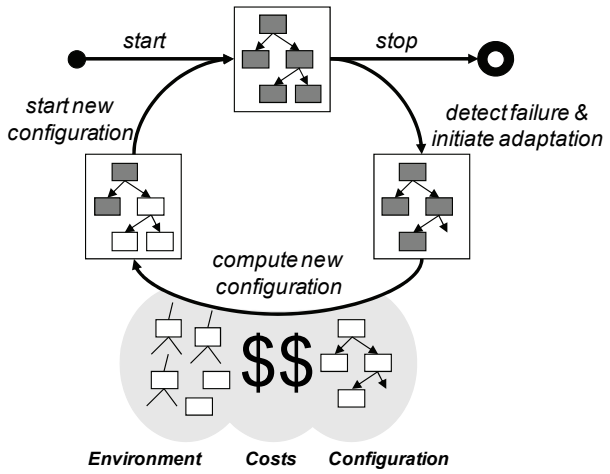

Fig. 2. Adaptation Process

Following this process, the system must ensure that invalid configurations can be detected, and it must enable the specification of feasible adaptations. In Section III A, we discuss the necessary refinement of the PCOM contract model to facilitate fully automatic adaptations. If an invalid configuration has been detected, a new valid configuration must be computed. As this process may require some time, we extend the component lifecycle to notify running components about the fact that the current configuration is temporarily invalid. Finally, as soon as an adequate adaptation has been computed, it must be applied to the existing configuration. For stateless components, this is a fairly straight-forward task. We need to stop all components that are no longer used and we start all new components. For stateful components, however, we additionally need to use the existing replacement mechanisms described in [12]. Due to space limitations, we do not discuss further details of this aspect.

A. *Refined Contract Model*

To facilitate parameterization, PCOM supports runtime changes to contracts. Since such changes are triggered programmatically, the knowledge about possible parameterizations is implicitly contained in the program logic of the component. In order to support adaptation by parameterization, the algorithm that computes a new configuration requires explicit knowledge about possible parameterizations. Thus, we extended the component model to support the specification of multiple optional contracts used to satisfy a requirement.

Since we do not want to start a global adaptation for every single fluctuation, we additionally define thresholds in which minor fluctuations can be compensated. To this end, we express requirements as fixed multi-dimensional ranges and offers as a dynamic multi-dimensional point. Thus, the system can detect that a changed offer no longer satisfies the corresponding requirement by determining that the point lies outside of the range. If this happens, an adaptation takes place.

B. *Extended Component Lifecycle*

If a configuration becomes invalid due to an unavailable component or an unmatched requirement, the component container that detects the invalid configuration signals this to the container hosting the application anchor. This container then initiates the computation of a new configuration, ensuring that simultaneously occurring signals will only initiate a single computation.

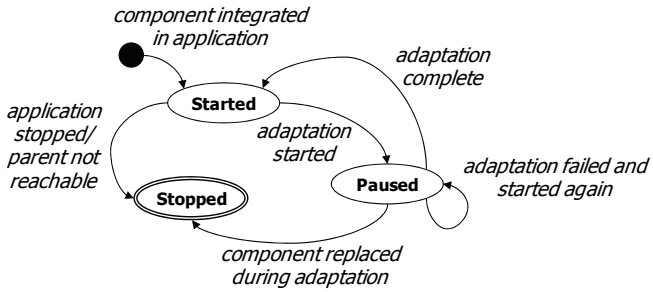

Fig. 3. Extended Component Lifecycle

Since computing a new configuration in the presence of scarce resources is a time-consuming procedure that might require multiple seconds, we additionally notify the remainder of components reachable from the application anchor about the invalid configuration. To do this, we introduce a PAUSED state into the component lifecycle. The PAUSED state is recursively triggered by the containers that host the application components. After the new configuration has been computed, all components that are still a part of the application are started again, and components that are no longer needed are stopped. The corresponding overall component lifecycle is depicted in Figure 3.

IV. Automatic Adaptation

When a configuration of a currently executed application becomes invalid, a new valid one must be computed. Under the

assumption that all valid configurations are equally well suited, this problem corresponds to the initial configuration problem.

As described in [11], this problem is defined as finding a tree of contracts for component instances starting from the application anchor whose leaf nodes do not require further instances and whose resource requirements can be met by the containers that host them. For an algorithm solving this problem, we have identified the following requirements:

- **Completeness**: It should find a valid configuration if one exists, and it should be able to determine whether a valid configuration does exist.
- **Efficiency**: It should minimize the delay for finding a valid configuration, since a user might be waiting for the application to be started.
- **Distribution**: It should be able to work in all environments without requiring a resource-rich device.
- **Resilience**: It should be resilient to failures that occur during the configuration process.

Clearly, from a high-level perspective these requirements remain valid for adaptation algorithms as well. Yet, the assumption that all possible valid configurations are equally well-suited for adaptation does not hold in general.

Usually, when an application needs to be adapted, there is still an invalid configuration available. Clearly, this configuration might lack some necessary component instances or it might require more than the available resources. However, if the adaptation algorithm does not take this configuration into account, the new configuration can easily lead to the replacement of a number of sub-trees that carry application-specific state. If this happens, the adaptation delay will be increased by the time required to restore the state of the replaced component instances. Depending on the application and the environment, this restoration delay might well exceed the delay caused by the algorithm itself.

Thus, in order to achieve the overall goal of minimizing the adaptation delay, an adaptation algorithm must minimize the sum of its execution delay and the resulting restoration delay. Intuitively, the execution delay and the restoration delay cannot be minimized independently since finding a configuration with a lower restoration delay will lead to higher execution delays. Therefore, in order to optimize the restoration delay, we rely on a heuristic that only imposes a minimal overhead on the execution delay.

In the following, we first propose a cost model to capture the restoration delay in an abstract manner. Based on the modeled restoration cost, we then describe our heuristic that minimizes these costs.

A. Cost Model

For our cost model, we first need to define the notion of a replaced component instance. For this, we need to consider that PCOM always replaces complete sub-trees in order to enable the consistent restoration of their state. This means that PCOM does not reuse component instances whose ancestor has been replaced. Thus, we can define the actually replaced instances as the set of the topmost instances of a configuration that have been removed. To do this, we define:

$$C_{config} = (I_{config}, D_{config}) \tag{1}$$

as a configuration consisting of the component instances $c_i \in I_{config}$ and the (directed) dependencies $(c_i, c_j) \in D_{config}$. Furthermore, we define:

$$Parent(c_i, C_{config}) = \{c_j \mid (c_j, c_i) \in D_{config}\} \tag{2}$$

with ***Ancestor*** as the transitive closure of ***Parent***. Thus, for an existing configuration C_{old} and a new configuration C_{new}, we get all removed instances as:

$$I_{removed} = I_{old} - I_{new} \tag{3}$$

and the set of the topmost removed instances as:

$$\begin{aligned} I_{replaced} = \{c_i \mid c_i \in I_{removed} \wedge \\ \neg\exists c_j : (c_j \in I_{removed} \wedge c_j \in Ancestor(c_i, C_{old})\} \end{aligned} \tag{4}$$

The amount of state that must be restored is given by the sum of the state held by the component instances in $I_{replaced}$ and the state of their recursively bound instances. If we define the amount of state locally held by an instance c_i as $S_{local}(c_i)$, then the state that must be restored for the replacement of an instance c_i with n child instances $c_{i,1} \ldots c_{i,n}$ is given as:

$$S_{total}(c_i) = S_{local}(c_i) + \sum_{j=1}^{n} S_{total}(c_{i,j}) \tag{5}$$

and the overall amount of state that needs to be restored for a new configuration is given by the sum:

$$S = \sum_{c_i \in I_{replaced}} S_{total}(c_i) \tag{6}$$

In general, the resulting restoration delay for a replaced instance depends on different factors. First of all, it depends on the amount of time required to transmit the state over the network. Secondly, it depends on the time required by the new instances to restore their internal state. Both delays basically depend on the size of the state which is captured in our model. We neglect any differences in the mechanisms by which different components restore the state internally as these will most likely not result in significantly different delays.

In summary, we can estimate the resulting restoration delay by computing the overall state that must be restored due to replacement of instances. Our optimization is based on the corresponding restoration cost.

B. Approach

To minimize the overall adaptation delay, we modify the algorithm that we use to compute the initial configuration in such a way that we do not increase its execution delay. Our configuration algorithm is based on Asynchronous Backtracking [23] which is a complete, distributed, and asynchronous algorithm. As discussed in [11], the algorithm works well for many typical problem instances. However, due to the very nature of the overall problem, it exhibits an exponential worst-case complexity.

Even with the simplified cost model presented in the previous section, the problem of finding a valid configuration with minimal restoration delay does not lie in NP. Thus, a complete optimization algorithm will need to explore a large number of possible configurations in many cases. Even in resource-rich environments, such an approach would inevitably lead to intolerable execution delays.

To reduce the restoration delay without increasing the execution delay of the algorithm, we add two heuristic extensions. Both extensions rely on a gradient descent technique and, thus, they are sensitive to the chosen starting point. In order to mitigate this, we propose to execute the overall algorithm multiple times using a randomized starting point in cases where the restoration delay is comparatively high and the already experienced execution time is low. This incremental approach has the additional benefit of quickly producing a valid configuration. This configuration may be used, or, if there is enough time, a better one can be found.

C. Configuration

To explain the heuristics, we briefly describe the mapping of configuration to a Distributed Constraint Satisfaction Problem (DCSP), we provide an overview of some relevant aspects of Asynchronous Backtracking (ABT) and we discuss an example of how it can be used for initial configuration. A DCSP is defined by a set of variables with finite domains that are distributed across a number of agents and a set of constraints between the values of variables. The goal is to find an assignment for all variables that does not conflict with the constraints.

To solve a DCSP with ABT, the variables have to be totally ordered by some priority criteria. If two variables share a constraint, ABT uses the variable with the lower priority to check whether the constraint occurs. To do that, a high priority variable ***a*** that shares a constraint with some low priority variable ***b*** sends its value assignments to ***b***. To model this, ABT relies on directional links. As a result, all variables that share constraints must be linked accordingly before the algorithm can be executed. At runtime all variables assign some value in parallel and send their assignments across their links. If some variable receives a value, it determines whether the current assignment is still valid (i.e. does not conflict with some constraint). If that is not the case, it tries to assign another value that does not conflict with its constraints. If that is not possible, it generates a backtracking message that contains the conflicting assignments that must be changed to resolve the problem. This message is sent to the conflicting variable with the lowest priority contained in the conflict set. If some variable receives a backtracking message, it request additional links from all variables in the conflict set, it checks whether the conflict is still present and if this is the case, it records the conflict as a new constraint. Thereafter, the variable tries to assign a valid value and the algorithm continues. ABT terminates if no valid assignment can be found or if all variables have stopped sending messages. In the latter case, the variables have found a valid solution.

To map the initial configuration to a DCSP, we interpret the component requirements defined in contracts (i.e. their dependencies) as variables. The contracts that can be used to satisfy them define their domain. The tree-based application structure and the resource conflicts between components define the constraints. To model that we are only interested in a partial solution of the DCSP, i.e. a solution for the variables that are recursively used by the application anchor, we add a pseudo value (***0***) to the domain of each variable. Using constraints that are built into the variables, we ensure that the pseudo value is chosen if and only if a certain sub-tree is not required. By virtually assigning the pseudo value to each variable, it is possible to construct the DCSP and the links required for ABT online without pre-computation. Thereby, variables and constraints are created on corresponding component containers when a contract is used by a dependency for the first time.

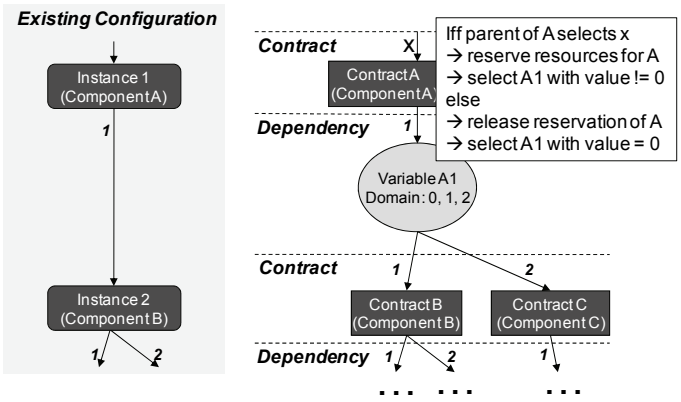

Fig. 4. DCSP Mapping

An example for such a mapping is shown on the right side of Figure 4: There are 3 contracts (***A***, ***B***, ***C***). The contract ***A*** is used if the value ***X*** is assigned to its parent variable. It declares one dependency (***A1***) that can be satisfied with either contract ***B*** or contract ***C***. When ***A*** is selected for the first time, the algorithm creates the variable ***A1*** for the dependency of ***A***, and it initializes the domain of ***A1*** with ***0***, ***1***, and ***2*** for the pseudo value, ***B***, and ***C*** respectively. Then the algorithm creates the links for ***B*** and ***C*** with ***A1*** so that the dependencies of ***B*** and ***C*** can determine whether they are used or not. Using a number of built-in constraints that are created for each contract, the algorithm can ensure that the variable assignments and resource reservations are always performed properly.

As a last step to apply ABT to initial configuration, we need to establish a total ordering between variables. Since the online mapping created by the algorithm creates links from variables of parents to variables of children, the ordering must ensure that parent variables have a higher priority than their children. However, unrelated path of the tree can have an arbitrary ordering. In [11] we discuss how such an ordering can be created by assigning specific IDs to variables. For the sake of brevity, we refer to [11] for further details on the initial configuration.

D. Heuristic Extensions

In order to minimize the restoration delay, we add two additional heuristics to the algorithm. To do that, we extend the previously introduced mapping.

The first heuristic is a value-ordering for variable values that puts a preference on parameterization. This means that if a certain dependency can reuse a component instance of the configuration by selecting a certain set of contracts under a certain set of value assignments, it must first search through these assignments before it selects another one that replaces the instance.

The second heuristic makes use of the fact that the ordering between the variables of unrelated paths of the tree can be arbitrary. The general idea is that during backtracking, the algorithm should first change the variables that select instances causing low costs before it changes variables that select instances with high costs. Note that if two assignments conflict, the original algorithm changes the assignment of the variable with the lower priority first. Thus, by defining the priority on the basis of the state S_{total} carried by a potentially bound component instance for each variable, we can achieve the desired behavior.

While the first heuristic can be integrated in a straightforward manner, the integration of the second heuristic raises a problem. The correctness of ABT is based on the fact that the variable ordering is static. Thus, if a variable has assignments that can either reuse or replace an existing component instance, we could either assign the priority S_{total} to reflect the reused instance or ***0*** to reflect the replaced instance, but not both. An example for such a variable can be seen in Figure 4. Under the assumption that the usage of contract ***A*** allows PCOM to reuse the component instance ***1***, selecting contract ***B*** would reuse component instance ***2***. Selecting contract ***C*** instead would require the creation of a new instance of type ***C*** and the restoration of the state of instance ***2*** in the new instance.

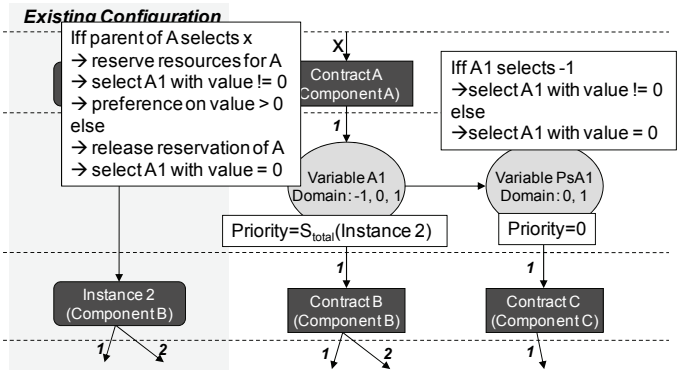

Fig. 5. Extended Mapping

Both options are suboptimal: If the algorithm already detected that the existing instance cannot be used, e.g. due to a resource conflict, changing the new instance does not increase the cost. Thus, if a newly bound instance conflicts with other parts of the application, it is more cost-efficient to change it before even more parts of the application are replaced. To avoid this problem, we split such variables into two variables, and we partition the possible value assignments into assignments that allow reuse and assignments that do not allow reuse. Thus, the algorithm can assign the proper static priorities at runtime. However, we additionally need to ensure that the algorithm does never select a contract for both variables. Therefore, we must introduce an additional pseudo value (***-1***) and a constraint that enforces this.

As an example for this, consider the DCSP fragment shown in Figure 4. When the algorithm initializes the domain of ***A1***, it detects that there are assignments that can reuse the existing instance (***A1=1***) and assignments that would replace the instance (***A1=2***). Thus, it splits the variable and creates a new pseudo variable ***PsA1***. The domain of the variable ***A1*** consists of the assignments that reuse the instance, including the pseudo values ***0*** (i.e. sub-tree not required) and ***-1*** (i.e. reuse not possible but sub-tree required). The domain of the variable ***PsA1*** is constructed from the assignments that do not reuse the instance and the pseudo value ***0*** (sub-tree not required or reuse possible). The algorithm adds a built-in constraint that models the fact that ***PsA1*** must only select a value if ***A1*** must be assigned but cannot reuse an existing instance (***A1=-1***). Since ***PsA1*** now shares a constraint with ***A1***, the algorithm adds a link from ***A1*** to ***PsA1***. As a final step, the algorithm must assign priorities to the variables. This can now be done statically. The priority of ***A1*** is initialized with S_{total}***(instance 1)*** since changing a selection (>0) could add these costs. The priority of ***PsA1*** is ***0*** since changing a selection (>0) would never add costs. The result of this procedure is shown in Figure 5.

In order to consider the priorities during backtracking, we need to replace the existing ***ID***-based variable ordering. As discussed earlier, the online DCSP mapping performed by the algorithm requires that a parent variable has a higher priority than its children. The definition of S_{total} already ensures that a parent has at least the same S_{total} than its child with the highest S_{total}. Thus, a parent will never have a lower priority. Yet, S_{total} does not guarantee a total ordering. In order to create such an ordering, we can combine the priority-based partial ordering resulting from S_{total} with the lexicographic ***ID***-based total ordering introduced in [11]. For two variables ***A*** and ***B,*** we define:

$$A < B \Leftrightarrow (priority(A) < priority(B)) \vee (priority(A) == priority(B) \wedge ID(A) < ID(B)) \quad (7)$$

Since the extensions require S_{total}, the algorithm needs to compute it for each existing instance. This can be done with a wave of messages sent through the configuration. Such a wave is already sent in order to toggle the PAUSED state of the instances. Thus, these messages can be used for piggy-backing cost information. This ensures that each component container knows S_{total} of the instances it hosts. Furthermore, the algorithm needs to ensure that S_{total} is always available on component containers that require it for a comparison. This can be

done by including S_{total} in all update and backtracking messages for all contained variables.

V. Evaluation

To evaluate the approach, we have measured the time required to pause an application, to initialize the cost model, and the time required to start a new configuration that solely changes the parameterization for varying application sizes. Note that the proposed adaptation heuristic does not impose any other additional execution delays. Thus, apart from these overheads, the measurements and simulations discussed in [11] remain valid and, therefore, we do not discuss them in this paper.

The measurements shown in Figure 6 have been gathered using 4 MDA Pro devices connected via IEEE802.11 on which we placed 1-12 components that form a binary application tree. The figure shows the average delay and the deviation of 500 runs per value. As indicated by the step increases at 2, 4, and 8, the overhead depends mainly on the height of the tree and on the latency for performing a remote call.

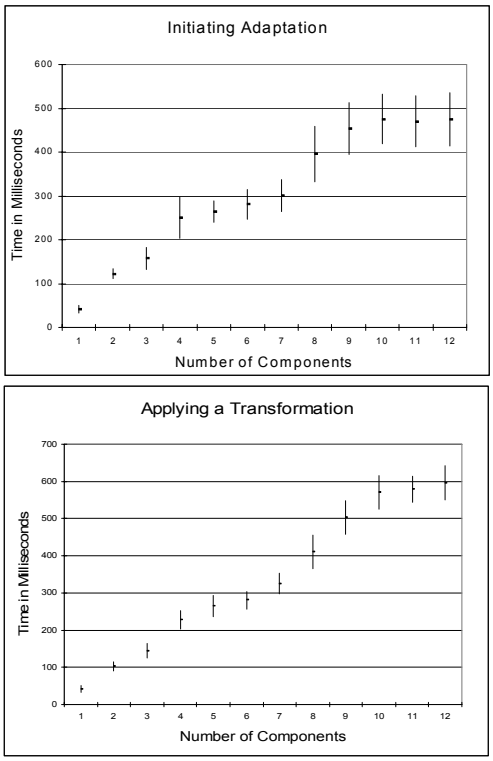

Fig. 6. Pausing (top) and Starting (bottom) a Configuration with Varying Number of Components

To measure the effects of the heuristics on the restoration delay (i.e. the costs S), we have implemented them using an event-discrete simulator, and we performed a variety of simulations. The simulation scenarios were set up using the following procedure:

We create a binary tree consisting of 15 contracts that originate from different components. This tree constitutes the running application. The cost S_{local} is randomly initialized using a standard distribution with an average of 10 and a deviation of 10. We use an abstract metric for the state size since we are only interested in the relative differences. The assumption here is that most components will carry a similarly low amount of state, yet, there are some components that carry high amounts of state information. In order to create alternative configurations, we randomly pick d sub-trees of the application and for each, we create a additional sub-tree that can be used as its replacement. For each contract of the new sub-tree, we decide with a probability p, whether the contract is a parameterization of the corresponding existing component (i.e. it can be reused). If it is not a parameterization, we create a new component for the contract. The resource requirements are initialized randomly such that each component requires 1 and 10 instances of a single abstract resource type. Then we create 4 containers on which we place the components randomly. Each container provides 30 resource instances. Finally, we increase the resource requirements of one of the contracts of the original application to 31 to simulate that the chosen configuration can no longer be executed.

To measure the quality of a solution found by the heuristic, we compute the distance of a solution cost (see definition of S in Section IV A) from the minimal solution cost achievable in each scenario, and we normalize the distance using the maximum and minimum solution costs of the scenario. As a result, each solution can be classified on a scale between 0 and 1 where 0 denotes the optimal and 1 denotes the worst solution with respect to costs. Using this quality measure, we now discuss a number of exemplary simulations. In order to get representative results despite the randomization, each experiment is based on 10000 simulations.

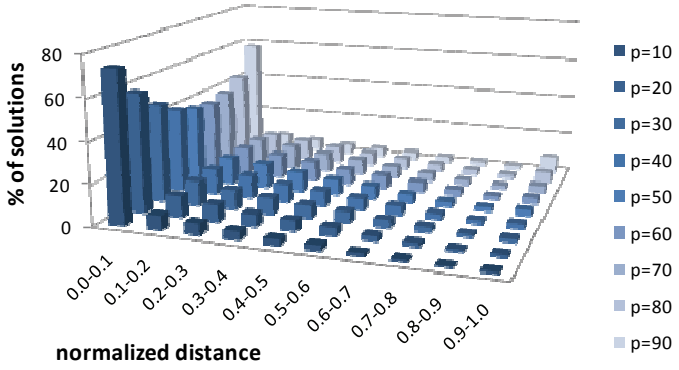

Fig.7. Single-Run Solution Quality for Increasing Parameterization Probability (d=20)

Figure 7 shows nine histograms of solutions grouped into 10 categories according to their solution quality for 20 duplicated sub-trees (***d=20***) and varying parameterization probabilities (***p=20 to 80***). Note that these histograms denote the solutions found after the first run of the algorithm. For a parameterization probability of 20%, the heuristics are able to find a solu-

tion that has a maximum normalized distance of 0.1 from the optimum in 58% of the simulations. Intuitively, if the probability is increased, the solution quality becomes worse up to a certain point where it increases again. The reason for this lies in the fact that if the probability is relatively low, there are only few parameterizations and, thus, the value-ordering heuristic works effectively. If the probability is high, most duplicated sub-trees will not inflict any costs and, thus, each parameterization leads to a similar (high) quality. If the parameterization probability lies around 50%, the scenario is likely to exhibit a number of parameterizations whose selection will indirectly introduce costs since they lead to the replacement of some child instance. Thus, the value-ordering becomes less effective.

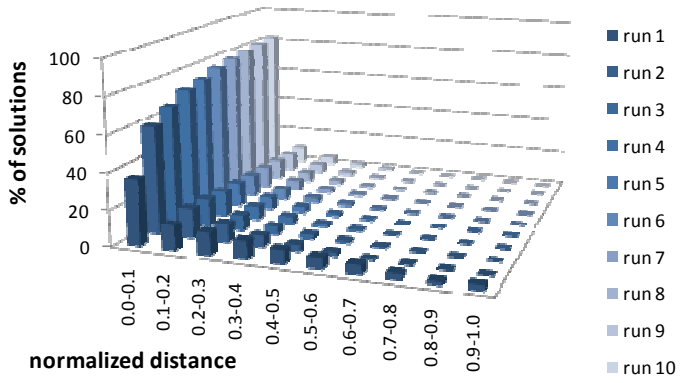

Fig.8. N-Run Solution Quality (d=20, p=60)

To improve the quality of the solutions, the algorithm is executed iteratively starting with a randomized variable assignment on each run, and the best configuration is used for the application. The resulting solution qualities after 1 to 10 runs for d=20 and p=60 are shown in Figure 8. The figure indicates that the quality can be increased significantly by running the algorithm a second time. Since the execution delay in these simulated scenarios is moderate (~100 messages/run), a second execution is likely to be worthwhile in cases where the first solution exhibits high costs.

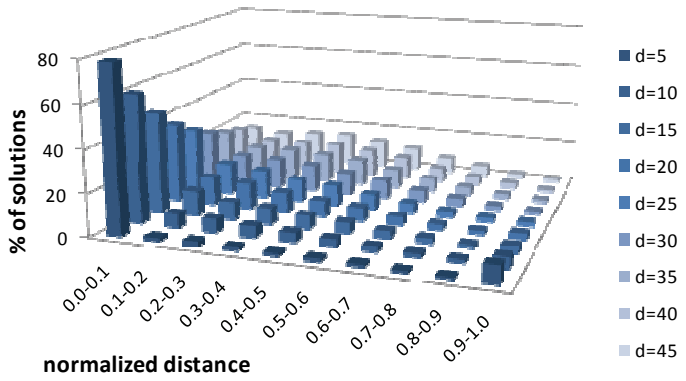

Fig.9. Single-Run Solution Quality for Increasing Duplicates (p=50)

Finally, in order to show the effects of a changing number of possible solutions, we have varied the number of duplicated sub-trees for a fixed parameterization probability ***(p=50)***. Figure 9 shows that the solution quality is reduced if the number of duplicated sub-trees is raised. This is most likely a result of the fact that the relative number of solutions with average cost increases disproportionately high. Again, this can be mitigated by executing multiple runs. However, in real world scenarios, the number of parameterizations supported by one component will be limited since each parameterization needs to be programmed and tested.

VI. Related Work

The ability to adapt to various environmental conditions is one of the key requirements for pervasive applications and there exists an extensive body of research in system support for adaptive applications.

In environments where changes occur infrequently, an application can be adapted manually [14], [5]. When changes occur frequently this is not a viable option.

In smart environments, adaptation is typically coordinated by a central entity that is hosted on a resource-rich device. In GAIA [20] and AURA [7], for instance, adaptation is performed by a coordinator that has a global knowledge of the services available in the environment. With PCOM we focus on environments where the availability of a resource-rich device cannot be guaranteed. To this end, the presented approach is fully decentralized.

Similarly, in GAIA the notion of adaptation differs from the one discussed in this paper. If an application is moved from one environment to another, this is a special instance of initial configuration. Adaptation in the same environment is initiated proactively by the user and not reactively [20]. The approach towards fault tolerance presented in [3] uses heartbeats to detect device failures. In addition to that PCOM uses contracts to model invalid configurations resulting from fluctuating resources.

Infrastructures like iRos [18] are based on the idea of loosely coupled applications. Such systems do not provide strong guarantees with respect to composition. This makes them hard to program in cases where coordination is required.

A number of systems target at application offloading in order to utilize remote resources [17], [8]. These systems automatically distribute an application. Typically, they do not provide means to deal with device failures which limits their applicability.

Odyssey [16] provides adaptation support for applications that access remote information. The system monitors the properties of the network between a client and a server and informs the client applications about changes. In PCOM such changes can be captured using contractual changes. In addition, PCOM can also switch between different configurations of a server.

Resource-aware application support has been investigated in the area of distributed multimedia applications [22], [1]. Mul-

timedia systems typically focus on wired networks and assume a static set of devices. Thus, they perform the initial configuration [10] and add network or media-based adaptation to compensate congestion or loss [15]. The approach in [9] proposes structural adaptation but it requires a central manager to compute and store alternative configurations.

The PCOM contract model exhibits a number of similarities with generic QoS specification languages like QML [6]. This paper not only discusses how to model contracts, it also discusses how they can be used to fully automate adaptation decisions.

Finally, there has been a lot of research on support for context-aware applications [4], [13]. Thereby, context is gathered to enable proactive adaptation [21]. While this type of adaptation is desirable, it is not sufficient in order to deal with changes that cannot be predicted reliably. Adaptation support for such changes is the focus of this paper.

VII. Conclusion

In this paper, we have presented an integrated approach for automatic reactive adaptation of pervasive applications. To enable full automation, we have introduced mechanisms that enable us to let the component system take care of all adaptation decisions. Furthermore, we have proposed a cost model and an optimization heuristic that are geared towards minimizing the adaptation delay in order to reduce the distraction experienced by users due the adaptations. The evaluation of our system indicates that in the majority of all cases it will be able to find a nearly optimal solution while adding only a minimal overhead for initializing the cost model. In cases where the solution quality is not satisfactory, a second randomized execution can significantly increase the quality.

We are currently implementing the heuristic as part of PCOM in order to test it with existing applications. Furthermore, we are working on strategies for deciding how many runs of the algorithm should be executed, based on prior solution costs and the execution times.

Acknowledgment

This work is partly funded by the German Research Foundation (DFG) as part of the Priority Programme 1140 – Middleware for Self-organizing Infrastructures in Networked Mobile Systems.

References

[1] C. Aurrecoechea, A. T. Campbell, L. Hauw, "A Survey of QoS Architectures", Multimedia Systems, 6(3), 138-151, 1998

[2] C. Becker, M. Handte, G. Schiele, K. Rothermel, "PCOM – A Component System for Pervasive Computing", IEEE 2nd Intl' Conference on Pervasive Computing and Communications (PERCOM'04), 67-76, 2004

[3] S. Chetan, A. Ranganathan, R. H. Campbel, "Towards Fault Tolerant Pervasive Computing", IEEE Technology and Society, 24(1), 38-44, 2005

[4] A. K. Dey, D. Salber, G. D. Abowd, "A Context-based Infrastructure for Smart Environments", 1st Intl' Workshop on Managing Interactions in Smart Environments (MANSE'99), 114-128, 1999

[5] W. K. Edwards, M. W. Newman, J. Sedivy, T. Smith, S. Izadi, "Challenge: Recombinant Computing and the Speakeasy Approach", 8th ACM Intl' Conference on Mobile Computing and Networking (MobiCom 2002), 23-28, 2002

[6] S. Forlund, J. Koistinen, "Quality of Service Specification in Distributed Object Systems Design", 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS), Santa Fe, New Mexico, April 27-30, 1998

[7] D. Garlan, D. Siewiorek, A. Smailagic, P. Steenkiste, "Project Aura: Towards Distraction-Free Pervasive Computing", IEEE Pervasive Computing, 1(2), 22-31, April 2002

[8] X. Gu, K. Nahrstedt, A. Messer, I. Greenberg, D. Milojicic, "Adaptive offloading inference for delivering applications in pervasive computing environments", IEEE 3rd Intl' Conference on Pervasive Computing and Communications (PERCOM'03), 107-114, 2003

[9] A. Hafid, G. Bochmann, "Quality of Service Adaptation in Distributed Multimedia Applications", Multimedia Systems, 6 (5), 299-315, 1998

[10] A. Hagin, G. Dermler, K. Rothermel, G. Shchemelev, "Distributed Multimedia Application Configuration Management", IEEE Transactions on Parallel and Distributed Systems, 11(7), 669-682, July 2000

[11] M. Handte, C. Becker, K. Rothermel, "Peer-based Automatic Configuration of Pervasive Applications", Journal on Pervasive Computing and Communications (JPCC), 1(4), 251-264, December 2005

[12] M. Handte, G. Schiele, S. Urbanski, C. Becker, "Adaptation Support for Stateful Components in PCOM", Workshop on Software Architectures for Self-Organization: Beyond Ad-Hoc Networking at Pervasive, 2005

[13] F. Hohl, U. Kubach, A. Leonhardi, K. Rothermel, M. Schwehm, "Next Century Challenges: Nexus – An Open Global Infrastructure for Spatial-Aware Applications", 5th Intl' Conference on Mobile Computing and Networking (MobiCom'99), 249-255, Seattle, USA, 1999

[14] J. Humble, A. Crabtree, T. Hemmings, K.-P. Akesson, B. Koleva, T. Rodden, P. Hansson, "Playing with the Bits User-Configuration of Ubiquitous Domestic Environments", 5th Intl' Conference on Ubiquitous Computing (UBICOMP 2003), 256-263, 2003

[15] K. Nahrstedt, R. Steinmetz, "Resource Management in Multimedia Networked System", Handbook of Multimedia Networking, 153-162, LNCS 1209, 1997

[16] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, K. R. Walker, "Agile Application-Aware Adaptation for Mobility", 16th ACM Symposium on Operating Systems Principles, 276-287, 1997

[17] S. Ou, K. Yang, A. Liotta, "An adaptive multi-constraint partitioning algorithm for offloading in pervasive systems", IEEE 4th Intl' Conference on Pervasive Computing and Communications (PERCOM'06), 116-125, 2006

[18] S. R. Ponnekanti, B. Johanson, E. Kiciman, A. Fox, "Portability, extensibility and robustness in iROS", IEEE 1st Intl' Conference on Pervasive Computing and Communications (PERCOM'03), 11-19, 2003

[19] M. Roman, C. K. Hess, R. Cerqueira, A. Ranganathan, R. Campbell, K. Nahrstedt, "Gaia: A Middleware Infrastructure to Enable Active Spaces", IEEE Pervasive Computing, 74-83, October-December 2002

[20] M. Roman, H. Ho, R. Campbell, "Application Mobility in Active Spaces", 1st International Conference on Mobile and Ubiquitous Multimedia, 2002

[21] B. Schilit, N. Adams, R. Want, "Context-aware Computing Applications", IEEE Workshop on Mobile Computing Systems and Applications, 85-90, 1994

[22] A. Vogel, B. Kerherve, G. v. Bochmann, J. Gecsei, "Distributed Multimedia and QoS: A Survey", IEEE Multimedia, 2(2), 10-19, 1995

[23] M. Yokoo, E. Durfee, T. Ishida, K. Kuwabara, "The Distributed Constraint Satisfaction Problem: Formalization and Algorithms", IEEE Transactions on Knowledge and Data Engineering, 10(5), 673-685, 1998