Customizable Pervasive Applications

Torben Weis, Marcus Handte, Mirko Knoll, Christian Becker [1]
Institute of Parallel and Distributes Systems, Universität Stuttgart, Germany
firstname.lastname@informatik.uni-stuttgart.de

Abstract

Human behavior and housing resist every standardization effort. Many aspects such as different technical equipment, furniture, and usage patterns make our surroundings as individual as ourselves. Thus, the personalization of pervasive applications is a fundamental requirement. To enable the development of custom pervasive applications, we propose a software development process. This process is based on the successful process for modern desktop applications. There, developers create extensible applications and components. Customizers use the resulting artifacts to develop custom applications. Finally, users configure applications to their individual needs by adjusting predefined settings. To adopt this process for Pervasive Computing, we present a component system for developers, a graphical toolkit for customizers, and self-configuration algorithms to ease the deployment.

1. Introduction

Pervasive Computing (PerCom) envisions user-centric support for tasks that go beyond the desktop. Computerized everyday objects that are embedded into the physical environment of users interact in a coordinated fashion to ease daily tasks. It has been well recognized that this vision eventually leads to heterogeneous environments and depends heavily on the individual preferences of users. As a result, it is unlikely that a single entity will be able to develop one-size-fits-all applications that match both, heterogeneous environments and heterogeneous user requirements.

One of the success factors of desktop applications such as office or creativity suites is the fact that they are highly customizable. Apart from the settings that can be adjusted by end-users, customizers can script additional functionality such as data import filters, macros, or specialized user interfaces. Since these applications are typically built upon extensibility frameworks, customizers can also reuse the provided features as part of their own applications. As a result, commercial off-the-shelf (COTS) desktop applications are often used as a foundation for the cost-effective development of custom applications.

In this paper, we argue that the development of pervasive applications should follow the successful trail of desktop applications. The contribution of this paper is twofold. First, we present a development process for customizable component-based pervasive applications. Secondly, we present Nexel and PCOM, our integrated tool chain for PerCom that supports this process.

The remainder is structured as follows. Next, we describe our system model. In Section 3, we present the development process and our approach for providing tools. In Section 4 and 5, we describe our tool chain consisting of PCOM and Nexel. Section 6 discusses related work and Section 7 concludes the paper.

2. System Model

We envision future pervasive environments as spaces, e.g., rooms and buildings, enriched with appliances. Appliances, such as phones, TVs, fridges, cups, etc., are equipped with wireless communication technology and they export their specialized functionality through high-level interfaces. Note that today this assumption has already become reality for many appliances. By buying new appliances, the environments of users are enriched gradually. Pervasive applications combine the specific features of appliances to achieve a coordinated behavior desired by their users. However, individual preferences and different sets of appliances might require custom application logic.

[1] This work is partially funded by German Research Foundation (DFG) Priority Programme 1140 “Middleware for Self-organizing Infrastructures in Networked Mobile Systems”, by the DFG Excellence Center 627 “Nexus”, and by Microsoft Research Cambridge.

3. Development Process and Tool Support

As shown in Figure 1, we can distinguish two software architectures that both foster customizable applications. Architecture one is the equivalent to modern productivity tools. A software vendor or an open source project produces a "suite". This is a feature-rich piece of software covering the functionality that most users expect from a smart environment. This suite can be customized by adding customization components. Architecture two builds on coarse-grained components. As Figure 1 indicates, a person can compose custom applications from these components.

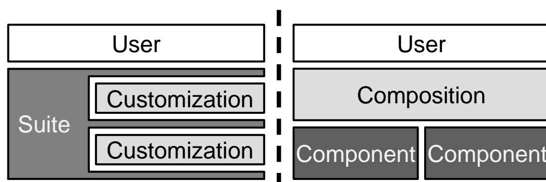

Figure 1 – Customization Architectures

Both architectures have in common that custom applications are built in three stages as shown in Figure 2. At the development stage, professional developers create commercial off-the-shelf suites, components or appliances for a large user base. At the customization stage, customizers adapt software to the special needs of rather small user groups. At the utilization stage, users deploy suites, components, appliances and customizations and expect that they cooperate seamlessly.

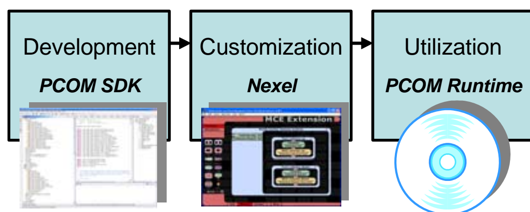

Figure 2 – Development Process and Tools

It is noteworthy that the groups of persons that adopt the roles of this process do not necessarily have to be disjoint. Instead, we believe that in the future especially technically interested persons will often act as customizers for their own pervasive applications. This clearly raises the requirement for appropriate development tools for each stage of the process.

As shown in Figure 2, our approach towards providing integrated tool support for this process is based on the combination of two software tools.

The integration of appliances as well as the development and utilization of applications is supported by our component system PCOM [2]. PCOM provides a component abstraction that can be used to develop pervasive applications. It relies on our middleware BASE that integrates various communication technologies. Thus, PCOM can act as bridge between appliances and since it performs the assembly of applications automatically at runtime, it enables users to execute applications without configuring them.

The customization of applications is supported by Nexel, our visual programming language. With Nexel, customizers can visually create PCOM components and applications. Using a plug-in framework, Nexel can utilize various appliances. This enables customizers that have only limited programming experience to visually orchestrate appliances. The generated artifacts can then be shared with other users.

In the following two sections, we provide an overview of the main concepts of PCOM and Nexel. To demonstrate customization of applications, our description is based on simple exemplary applications.

4. PCOM

In PCOM [2], applications are trees of components that are potentially distributed across devices. Each component provides certain functionality to its parent by relying on the functionalities of its children. The exact composition is automatically determined and maintained at runtime by the system. To do this, each component is equipped with a contract that declares its dependencies towards the local execution environment and other components. Using these contracts, a component container on each device ensures that the requirements of all used component are fulfilled at any point in time. This kind of self-configuration [8] enables PCOM to execute applications in different environments without any manual setup or intervention.

To show how this system fits into the development process, consider the volume control application shown in Figure 3. The application enables a user to control the volume of a media player using a phone.

In this example, the application anchor, i.e., the root of the tree, is equipped with a contract that declares dependencies to two components (`mediaPlayer` and `mobilePhone`). Components that can be bound to the dependencies must provide certain events and interfaces (a), (b). Whenever an instance of a `VolumeControl` is started, the component container performs automatic binding using matching components. Figure 3 shows one component for each dependency that can fulfill the requirements (c), (d).

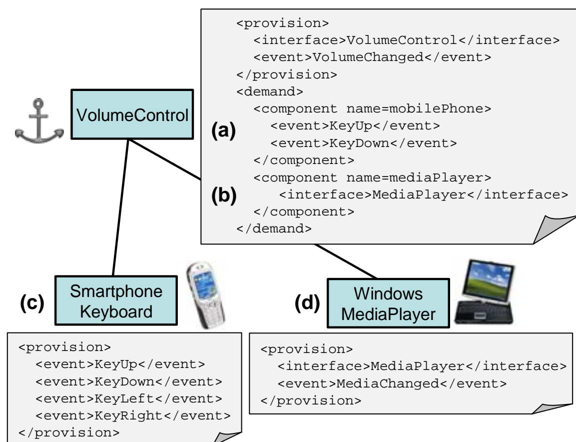

Figure 3 – Volume Control Application

For the sake of simplicity, our example is purely based on syntactic contracts and omits details that have been described in previous publications [2], [8]. [4], for instance, identifies four levels of contracts: syntactic, behavioral, synchronization and QoS contracts. PCOM contracts can model QoS dimensions and assignments as well. Furthermore, they can be used to define resource requirements towards their executing container. For our later description of Nexel, it is important to mention that the matching of contracts is not tied to the type system of the underlying programming language. Thus, the interface and events specified in contracts can be arbitrary identifiers.

Using PCOM, developers can integrate appliances as components with specified interfaces. The customization of applications is mainly supported by the late binding performed by the PCOM container and the recursive nature applications. As applications are composed at runtime, all parts of the application can be extended (e.g., through adapter components that provide additional functionality or enable the integration of new appliances). Furthermore, since an application anchor itself is just a component, customizers can combine existing applications and components by building new components that rely on their interfaces.

To deploy an application, a user must download its components onto the devices in the target environment. As PCOM has been specifically designed to automatically configure applications at runtime, usage does not require any configuration. Using a graphical user interface, a user simply starts an application anchor and the system will run the component as long as all required components are available. If an appliance becomes unavailable, PCOM will automatically try to find an adequate replacement. To enable this, we have developed an initial set of algorithms [8] and mechanisms. These algorithms encompass greedy-based heuristics as well as complete solutions based on asynchronous backtracking. However, regarding automatic configuration there are still interesting research questions that have not been solved so far.

5. Nexel

Using PCOM developers can create commercial off-the-shelf (COTS) components which automatically orchestrate themselves. Thus, as a user you download new components and the environment will integrate them. However, quite often some glue logic is missing to combine a set of components to a useful application. Imagine that your phone and your media player are already PCOM-enabled and you want to control the volume of the player via the phone. All you need is a component that ties together the functionalities.

To simplify the development of such small but useful pervasive applications, we have developed a graphical programming language called Nexel. We believe that such a language can do to pervasive applications what VisualBasic & friends did for simple GUI and database applications: Nexel allows casual hobby programmers to produce PCOM components and to share them with others. In the following, we illustrate its basic features by implementing previous example.

First, we must select the components that we want to work with. Nexel itself has no built-in support for any special component. Instead, Nexel can be extended via plug-ins. By dragging a component on the window, Nexel adds component-specific commands and events to its sidebar. In our case, the smartphone emits `Up key` and `Down key` events, while the media player features a `MP change volume` command.

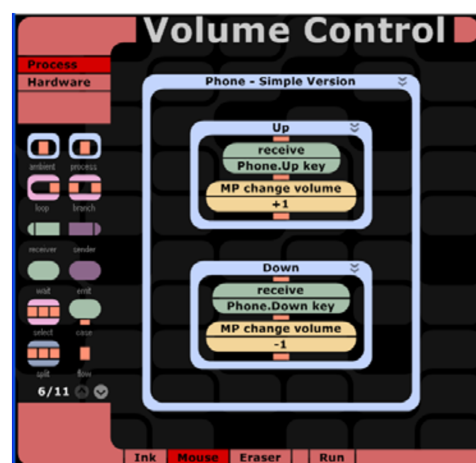

Figure 4 – Volume Control with Plug-Ins

Secondly, we create a component called “Phone – Simple Version” (see Figure 4) by dragging the component symbol from the sidebar. Furthermore, we add two processes: one for increasing and one for decreasing the volume. In an endless loop, the process waits for the `Up/Down key` event and executes the `MP change volume` command. One key advantage of Nexel is that these tasks can be achieved with simple Drag & Drop. There is no need to understand PCOM.

In step three, Nexel generates a PCOM component with two required contracts: one for the phone and one for the media player. By pressing the “Run” button at the bottom of the screen we can deploy the component and use the resulting application from within PCOM.

5.1. Implicit Component Contracts

The example in Figure 4 is mapped to a PCOM component with two demanded contracts, one for each used component. The phone plug-in tells Nexel that it provides the following contract:

```
<provision>
  <event>Phone.Up Key_int</event>
  <event>Phone.Down Key_int</event>
  <event>Phone.Left Key_int</event>
  <event>Phone.Right Key_int</event>
</provision>
```

Nexel analyzes the application and realizes that only the `Up Key` and `Down Key` events are used. Thus, it demands the following contract:

```
<demand>
  <component name=”Phone”>
    <event>Phone.Up Key_int</event>
    <event>Phone.Down Key_int</event>
  </component>
</demand>
```

The same happens with the media player. It provisions the following PCOM contract:

```
<provision>
   <interface>Volume.Up_int</interface>
   <interface>Volume.Down_int</interface>
   <interface>IMediaPlayer</interface>
</provision>
```

The provisioned contract states that the media player can receive events (in contrast to the phone which emits events) for volume up/down and the expected type is int. Thus, Nexel uses a concept known from C++ linkers: name mangling. It mangles the namespace (`Volume`) with the name of the identifier (`Up`) and the type (`int`) into one unique identifier. To PCOM this does not matter, since a PCOM interface is just an identifier. To avoid that two independent developers create two identifiers with the same name but different semantics, Nexel could as well use strong names as in .NET. To continue with our example, Nexel will generate a second demanded contract:

```
<demand>
 <component name=”player”>
  <interface>Volume.Up_int</interface>
  <interface>Volume.Down_int</interface>
 </component>
</demand>
```

The contracts we have investigated so far use a concept known as Signals & Slots [13]. Signals & Slots are extensively used in KDE/Qt desktop applications to connect components. A signal allows a component to emit a named value. A slot receives and processes a named value. In our example, the phone emits an event of name `Phone.Left` and type `int`. This can be treated as a *signal*. The media player offers an interface that can receive an `int` under the name `Volume.Up`, i.e. it is a *slot*. As outlined in the next section, Nexel has built-in support for Signals & Slots.

However, the media player offers an additional interface: `IMediaPlayer`. This is not a mangled name and therefore neither a signal nor a slot. Instead, it is a more low-level but powerful Java interface. Using a plug-in, Nexel can utilize such interfaces. It is the task of the plug-in to extend Nexel with new commands (i.e., new GUI elements), that – when executed – call the low-level interface. However, without plug-in support, Nexel cannot make such calls. This dualism between Signals & Slots and low-level Java interfaces is comparable with the `IDispatch` interface in Microsoft COM. High-level languages talk to COM components only via `IDispatch`, although they can additionally offer more powerful interfaces for C.

Using contracts implicitly, Nexel developers do not get in touch with them. It happens all behind the scenes of our tool chain. However, this approach works only if somebody (i.e., the appliance vendor or a third party) provides a Nexel plug-in. In the next section we show how you can deal with contracts explicitly.

5.2. Explicit Component Contracts

Imagine that our media player offers Signals & Slots in its interface, but the media player vendor did not provide plug-in. In this case, we cannot utilize `IMediaPlayer`. However, we can still utilize Signals & Slots. To illustrate this concept, we implement the application of Figure 4 again using explicit contracts. The result is depicted in Figure 5. The major difference here is that Figure 5 does not require a plug-in. Instead, the Nexel component explicitly declares a demanded contract of name `Volume` with two signals named `Up` and `Down`. A signal is notated as:

Volume.Up: int

The corresponding slot is notated as:

Volume.Up: int

To increase/decrease the volume, the Nexel component posts a value ("1" in the example) to such a signal. Therefore, it uses the built-in `post` command. In Figure 4, we used the media player specific `MP change volume` command, but it is not available here, since we do not use a plug-in.

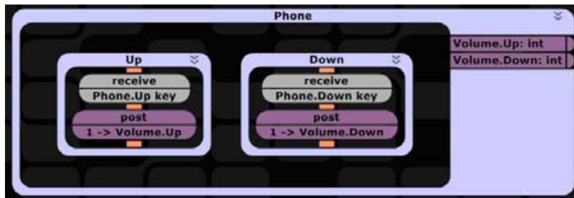

Figure 5 – Volume Control via Contracts

When Nexel maps the component in Figure 5 to PCOM, the result is exactly the same as in Figure 4. This dualism between implicit and explicit contracts exists for two reasons. First, implicit contracts via plug-ins ease the work of Nexel users. Second, using such plug-ins Nexel can use low-level Java interfaces like `IMediaPlayer`. The plug-ins just have to provide adequate commands. In contrast, Signals & Slots can always be used even if no plug-in is available.

5.3. Extensible Applications

Customization always requires two parties to cooperate. First, applications must be extensible. If everything is hardwired, we cannot customize anything reasonable. Second, customizers must extend applications by providing additional customization components.

The previous examples have shown how to build an extensible application in Nexel. Our application is not hardwired to any special component or device. It can cooperate with any component offering matching contracts. This way, our application is extensible. For example, someone could provide a component that implements the `Volume` contract and shows the current volume on the TV screen whenever the volume is changed. In the next section, we show how to extend our application to make it MediaCenter PC compatible.

5.4. Extending Applications

Getting COTS (commercial off-the-shelf) applications helps users to orchestrate most of their appliances but they will not satisfy all desires. Two concerns are most likely to arise. First, the contracts of COTS applications will not always match the contracts available in the environment. Secondly, users want to extend their applications to provide additional functionality.

The first issue can be solved by implementing component adapters that translate the provided functionality into the desired interface. This is analogous to the example shown in figure 5, except that the customizer has to model provided Signals & Slots by attaching them to the left side of their component.

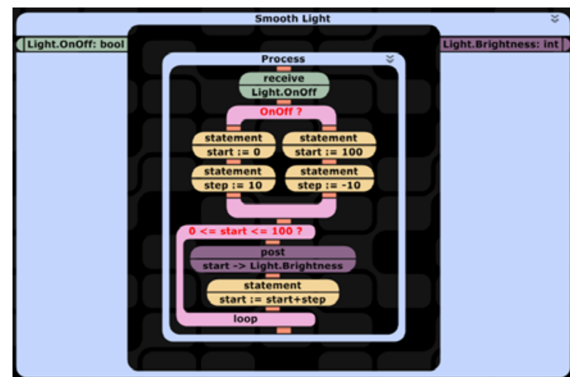

Figure 6 – Component for Light Dimming

The second problem can for instance be tackled by implementing intermediary components. Consider the following example: first, the user buys a basic starter set for controlling his home. Then as he gets more experienced, he is up to new challenges and needs add-ons. For instance, the user might want to change the behavior of the light switches. Instead of simply switching them on and off, he wants the lights to fade. To do this, we add a component in between the COTS application and the light component as shown in Figure 6. The new component provides the feature `Light.OnOff` and demands a component with brightness control, i.e. the `Light.Brightness` interface. If an application uses `Light.OnOff` then PCOM connects it to our `Smooth Light` component which is in turn connected to the real light component.

Thus, we extended an application with a feature. Whenever the lights switch, `Smooth Light` translates the command into a sequence of dimming steps.

6. Related Work

We presented a development process for customizable pervasive applications. Our way of developing applications has a number of similarities with the ideas presented in [1]. In contrast to this work, we have presented a tool chain that can be used to realize the ideas.

To the best of our knowledge, there is no other solution that covers all aspects discussed in this paper. Yet, there has been extensive research in specific sub-areas.

In the past, researchers have often focused on novel abstractions that ease the development of adaptive applications [6], [7]. The resulting infrastructures, however, do only provide limited support for customization. GAIA [11] has contributed profound knowledge regarding the development of system software for smart meeting rooms. To customize applications, GAIA applies a two-step mapping process in which an application description is mapped onto the devices of an environment. While this approach can be used to adapt an application to a certain environment, it cannot support the customizations supported by Nexel.

In the area of end-user programming, a number of tools have been developed. [10] is a tool that uses a story-board approach. In contrast to our approach, this tool does not support decoupled components that can be combined to new applications. [9] is an editor that allows users to connect different components. Yet, it does not allow non-linear control flows making it difficult to model complex application behavior. [3] presents a toolkit for end-user programming. The focus of their work lies on interacting with active environments. This allows the user to create rules by simply arranging tangibles on a floor plan. A drawback of this system is the dependency on special hardware as well as the fact that the user has to create unambiguous rules. Other rule-based systems like [12] and [5] offer the same features solely basing on different input methods. An important and fundamental difference between all these systems and our approach is the fact that we are not aiming at end-user programming for everyone. Instead, we envision the development of custom pervasive applications as a process where technically interested persons can develop customizations that can be used by others as well. This way we can satisfy a broad spectrum of user requirements without requiring that all persons create customizations.

7. Conclusion

The contribution of this paper is twofold. First, we presented a process that supports the cost-effective development of customized prervasive applications. This process is a middle course between two extremes: one-size-fits-all COTS applications and end-user programming for everyone. Secondly, we have presented a software development solution that supports developers, customizers and users. Our tool chain features a unique combination of component-based software, visual programming, and self-configuration.

In the future, we will put our focus on preferences that enable users to gain more control over the configuration process. Using preferences users will for example be able to state that they want a dimmer component for the bedroom but not for the kitchen. We believe that this will reduce the loss of control that is inherently associated with any kind of automation. Furthermore, we are investigating how we can utilize Nexel for rapid prototyping. This way, customizers as well as developers can benefit from the tool.

8. References

[1] Banavar, G., Beck, J., Gluzberg, E., Munson, J., Sussman, J., Zukowski, D., “Challenges: An Application Model for Pervasive Computing”, Intl’ Conf’ on Mobile Computing and Networking, 2000, pp. 266-274

[2] Becker, C., Handte, M., Schiele, G., Rothermel, K., „PCOM – A Component System for Pervasive Computing“, IEEE Intl’ Conf’ on Pervasive Computing and Communications, March 2004, pp. 67-77

[3] Beckmann, C., Dey, A., “SiteView: Tangibly Programming Active Environments with Predictive Visualization”, Intel Research, IRB-TR-03-025, 2003

[4] Beugnard, A., Jezequel, J., Plouzeau, N, Watkins, D., „Making Components Contract Aware“, IEEE Computer, vol. 32, no. 7, July 1999, pp. 38-45

[5] Dey, A., Hamid, R., Beckmann, C., Li, I., Hsu, D., “a CAPella: Programming by Demonstration of Context-Aware Applications”, ACM Conference on Human Factors in Computing Systems, 2004, pp. 33-40

[6] Garlan, D., Siewiorek, D., Smailagic, A., Steenkiste, P., “Towards Distraction-Free Pervasive Computting”, IEEE Pervasive Computing, vol. 1, no. 2, 2002, pp. 22-31

[7] Grimm, R., “One.World: Experiences with a Pervasive Computing Infrastructure”, IEEE Pervasive Computting, vol. 3, no. 3, July-September 2004, pp. 22-30

[8] Handte, M., Becker, C., Rothermel, K., „Peer-based Automatic Configuration of Pervasive Applications“, IEEE Intl’ Conference on Pervasive Services, 2005, pp. 249-260

[9] Humble, J., Crabtree, A., Hemmings, T., Akesson, K., Koleva, B., Rodden, T., Hansson, P., “Playing with the Bits User-configuration of Ubiquitous Domestic Environments”, Intl’ Conf’ on Ubiquitous Computing, 2003, pp. 256-263

[10] Li, Y., Hong, J., Landay, J., “Topiary: a tool for prototyping location-enhanced applications”, ACM Sym’ on User Interface Software and Technology, 2004, pp. 217-226

[11] Roman, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R. and Nahrstedt, K., “A Middleware Infrastructure for Active Spaces”, IEEE Pervasive Computing, vol. 1, no. 4, pp. 74-83, October-December 2002

[12] Sohn, T., Dey, A., „iCAP: An Informal Tool for Interactive Prototyping of Context-Aware Applications“, ACM Conf’ on Human Factors in Computing Systems, 2003, pp. 974-975

[13] Weis, T., Geihs, K., „Components on the Desktop“, IEEE Tools Europe, 2000, pp. 250-261