uDesign: End-User Design Applied to Monitoring and Control Applications for Smart Spaces

João Pedro Sousa,[†] Bradley Schmerl,[‡] Vahe Poladian,[‡] Alex Brodsky[†]
[†]*George Mason University*
Fairfax VA 22030 USA
{jpsousa, brodsky}@gmu.edu
[‡]*Carnegie Mellon University*
Pittsburgh PA 15213 USA
{schmerl, poladian}@cs.cmu.edu

Abstract

This paper introduces an architectural style for enabling end-users to quickly design and deploy software systems in domains characterized by highly personalized and dynamic requirements.

The style offers an intuitive metaphor based on boxes, pipes, and wires, but retains enough preciseness that systems can be automatically assembled and dynamically reconfigured based on uDesign descriptions. uDesign was primarily motivated and validated within monitoring and control applications for smart spaces, but we envision possible extensions to other domains.

Our contribution differs from early attempts at end-user programming by dealing with higher level software architectural abstractions rather than programming, and by addressing run-time descriptions rather than code structures.

The paper presents validation of uDesign along the following aspects: (a) expressiveness, by means of two case studies, one in health care, and one in home security, (b) soundness, by providing uDesign's formal semantics, and (c) implementability, by describing a mapping of uDesign to an existing software infrastructure: the Aura infrastructure.

1. Introduction

Easy assembly of software systems is increasingly important in domains such as assisted living and long term healthcare, smart homes, surveillance of public and private spaces, and emergency response. Such domains are characterized by (i) highly personalized requirements, for which generic one-size-fits-all software solutions are less than ideal; and by (ii) dynamic changes, both with respect to which devices are convenient to use and the requirements for the system.

For example, a doctor should be able to easily write a prescription for the healthcare features and behavior of an outpatient's home, much like medicine is prescribed today. The patient could tailor the prescribed behavior to suit personal and privacy preferences; for instance, by including family members as first line responders. Also, the components and behavior of the system might be adjusted over time, by healthcare professionals or the patient, to accommodate new devices and/or behaviors in response to the patient's progress.

Existing approaches focus on easy deployment of solutions by means of "friendly" programming environments (e.g., [1][3]) and by exploiting new technologies such as service-oriented computing and composition of web services.

Nevertheless, designing and assembling such systems remains a task that requires a fair amount of effort and programming skill. Deploying an application that used to take days or weeks for constructing the code from scratch may now be reduced to a few hours for a trained programmer. However, the skill and effort required for that is still beyond the capabilities and willingness of end-users.

This paper introduces an approach that allows end-users to assemble and evolve highly personalized software systems for monitoring and control in smart spaces. Ideally, such an approach is:

a) simple enough for end-users to manipulate with little initial training;
b) effective as far as the ratio between the recognized benefit and the effort spent; and,
c) precise enough to enable the automatic assembly of a running system based on a description provided by the end-user.

We hypothesize that an approach based on code structures and programming primitives is too fine grained and removed from the experience of end-users for achieving such goals. Instead, our work investigates whether combining the component and connector view of a system's architecture with activity-oriented computing (more below) results in a suitable foundation to address this problem.

The conceptual model that we propose uses a metaphor of boxes, pipes, and wires. This is similar to consumer electronics, where end-users may buy a number of devices and cables and try different assembly configurations having a basic knowledge of what travels on each cable, but without having to understand the corresponding electrical specifications.

The contribution of this paper is uDesign, an architectural style for describing systems of the class exemplified above, for which a formal semantics is defined, as well as a mapping to an existing software infrastructure. uDesign is an architectural style in the sense that it prescribes the kinds of components and connectors that can be used to assemble a system. It can be thought of as an extension of the pipe-and-filter style [10] where boxes are more general than filters of data, and a new kind of connector is made first class: wires for controlling the starting and stopping of activities in boxes.

In the remainder of this paper, Section 2 presents a brief rationale for the organization of uDesign and compares with related work. Prior work by the authors focused on the automatic assembly of systems given a specification of the available resources and of its required features [12]. Such required features are derived from representing user activities as first class constructs in software systems, giving rise to activity oriented computing [13]. High-level mechanisms for specifying the interconnection and coordination of the parts of a system have been lacking, though, and that is precisely the focus of this paper.

Section 3 leads the way for the presentation of the case studies by offering a description of uDesign's concepts at a level that would be appropriate for end-users.

Sections 4 and 5 present two examples, one in long-term healthcare, and another in home security and automation. Being able to understand case studies such as these after the informal introduction in Section 3 constitutes supporting evidence concerning the simplicity and effectiveness of uDesign: goals (a) and (b) above. However, fully validating these goals requires conducting user studies that present realistic problems to real users. For that, tools for editing uDesign must be brought to a level of maturity where they can be used by non-computer scientists. This is the object of ongoing and future work.

Concerning the preciseness of uDesign, goal (c), above, Section 6 enumerates its syntactic primitives and specifies their semantics using Zed [14], while Section 7 maps those primitives to an existing software infrastructure.

Section 8 concludes the paper and summarizes the main contributions and future work.

2. Approach

uDesign differs from other languages targeted at end-users in two fundamental aspects: it represents run-time structures rather than code structures, and it differs in the level of abstraction of such structures. Additionally, as frequently done in design disciplines, uDesign supports separable views of structure and behavior (e.g. [2]).

First, the boxes in uDesign correspond to running entities that are available to be incorporated in a system, rather than to classes or instance factories. Choosing the latter option would mean that end-users would have to create logical abstractions, i.e., programs or scripts, to control the creation, interconnection, and destruction of instances in the system.

In contrast, uDesign relies on discovery mechanisms to identify service instances that are available, and offers interactive primitives for end-users to integrate and interconnect those services into a system.

Second, there is a clear tradeoff between the detail that the user is asked to manipulate and the usability for a broad user base. The more detail, the more power the user has to construct complex behaviors, but more effort and training are required to use that detail. To help manage this tradeoff, a recent trend set by service-oriented computing is to have a separation of the roles of service supplier and service consumer. uDesign takes that trend one step further by supporting two groups of service consumers: domain specialists, such as doctors; and end users with a general education.

Services are required to work out of the box, with a default behavior, or possibly with a set of typical behavior templates. A general user should be able to make use of such services using the default behaviors or possibly recognizing abstract parameters or modes of operation, such as normal operation and emergency operation. Domain specialists, or technically bent users, would be able to understand and tailor those generic templates; for example, a doctor defining that the emergency mode corresponds to the heart rate exceeding 140 beats per minute (bmp) for a given patient, but only 120bpm for another patient.

2.1 Related work

A number of research projects, such as eHome [6][7], AMIGO [1], ETRI Open Home Framework [4], have addressed challenges in home task automation. The technical problems addressed by these projects include: device and software interoperability, deployment management, and installation-time configuration.

The eHome Systems project [6][7] addresses device interoperability, installation time configuration and deployment automation, with the focus of reducing the costs due to home automation product and service vendors. eHome's three-phase software process model: Specification, Configuration, and Deployment (SCD), logically parallels the task description and task configuration steps in the Aura Software Architecture, which forms the infrastructural basis for our current work. The eHome Configurator tool leverages the configurable features of the eHome platform and allows vendor technicians to easily tailor the installation to the needs of the client. Unlike the eHome project, that targets installation time configuration by vendors of software, uDesign targets everyday users and allows configuration after installation.

The Open Home Framework (OHF) developed by the ETRI institute [4] focuses on hardware, software, and protocol interoperability and integration issues. Having collaborated with the ETRI institute, we have discovered that the features offered by Aura, and uDesign specifically, are complementary to those provided by the OHF Home Server. Specifically, uDesign allows end users to define tasks for communication and notification tailored to a user's unique needs, preferences, and context.

The Amigo project has proposed a reference architecture [1] for networked home service automation. The key issue handled by the architecture is interoperability among different vendors of device and service providers. According to the project web site, the architecture will provide the following features: context awareness and notification, quality of service, user security and privacy. These features are not yet fully designed or documented. While the Amigo architecture provides some functionality similar to that of uDesign, the latter is targeted to the end-user for flexible runtime configuration, while Amigo does not offer such features.

Another domain where a service-oriented, diagrammatic approach to constructing activities is being investigated is in the domain of robotics. Microsoft Robotics Studio [5] uses a Visual Programming Language (VPL) as its main programming description. Users can drag and drop services into a diagram and connect them together. The graph then forms a dataflow-based program that is used to control a robot. The dataflow connections are strongly typed, and the realization of services can be chosen to be simulations or robot code. The approach is similar to uDesign. It does not allow resumption or suspension of activities as in uDesign, and mixes the structural and behavioral aspects of the dataflow. Furthermore, the target audience is robotics programmers, rather than end users.

There has been considerable work on Business Process Execution Language (BPEL) (e.g., see [8] for an overview and formal semantics), and Business Process Modeling Notation (BPMN) (e.g., see [15] for an overview and mapping to BPEL). BPEL is an executable business process language, serialized in XML, to support *programming in the large.* BPEL allows one to specify a business process behavior, both of a participant, and of a protocol with visible message interchange. BPEL's scope includes the description of process activities and their partial ordering, correlation of messages and process instances, and recovery behavior. While BPEL is a textual language (XML), BPMN is designed around a graphical notation, and can be used as a graphical interface for BPEL (although we are not aware of one-to-one mappings between BPEL and BPMN). While the motivation behind BPEL and BPMN is to allow the specification of executable processes by people who are not necessarily programmers, it still requires one to understand the level of abstraction that is beyond the capabilities of a typical end user, whereas, the level of abstraction for a typical end user is exactly the focus of this paper.

For monitoring and control applications that require persistent storage, there has been extensive work on *Active Databases* in the database community (e.g., see [9] for overview). Active Databases extend relational or object-oriented databases with Event-Condition-Action (ECA) rules. Each such rule is triggered when a designated event occurs, and then, if the condition in the rule is satisfied, an action is taken. Conditions may involve regular database queries, and actions may involve triggering other rules. Using the ECA paradigm within a database management system allows for standard database features, including atomicity, consistency, isolation and durability of transactions, which may be critical in many application domains. However, the level of abstraction in Active Databases, is that of SQL (or SQL-like) language, extended with triggers, which is not the level of abstraction that can be handled by typical end users.

3. Getting started with uDesign

This section introduces the concepts in uDesign at an intuitive level, illustrating the understanding that end-users need to have to create and tailor systems such as the ones presented in Sections 4 and 5. A technical overview of uDesign is presented in Section 6.

The three main constructs in uDesign are boxes, pipes, and wires. Boxes are the locus of computation, while pipes stream data among boxes. Wires control starting and stopping activities on boxes, as well as the flow of data on pipes, based on observed conditions.

To help manage visual clutter, uDesign defines three overlays: *structural, box behavior,* and *pipe behavior.* The **structural** overlay identifies the boxes, their properties and internal structure, and the piping of data among boxes. Boxes may be wrapped inside larger boxes, to allow scaling to more complex systems, or simply to hide details from other users.

Boxes correspond to entities of interest or their activities. For example, boxes may be associated with the TV set in the user's living room, with the living room as a whole, or with the user's activity of following a

TV show. Boxes may also be associated with software components, which like devices are viewed in the perspective of a concrete operating component that contributes to the system's function.

When a box is associated with a physical space or an object, such as a couch, what really happens in the system is that the box is realized as a combination of software and hardware that monitors and maybe controls the corresponding physical entity. Typically, such realization is provided as part of the entity: constructors will sell smart homes, and furniture stores will sell smart couches (or the means to make old couches smart.) It will be up to end-users to determine how smart objects can be assembled and reconfigured, via their corresponding boxes, to serve the users' needs.

Users and their activities may have associated boxes. Such boxes identify the properties of interest and clarify the user's role in achieving the system's intended function. Whether to represent a holistic view of a user or a specific view of the activities of concern is a decision for the end-user to take. In either case, smart spaces will be equipped with generic software components for modeling activities, and which may be associated with humans and their activities.

Boxes have inputs, which are entry points for data, and properties. Properties are any observable aspect of a box, such as the video output of a DVD player, whether it is powered up, or its location.

Data may be piped between any property of a box, a producer of data, and an input in a box, a consumer of data. Whenever a piece of data is available on the producer side, the pipe will transmit it towards the consumer side. uDesign tools check for type compatibility and disallow invalid piping, such as trying to pipe a video output to a textual input.

The **box behavior** and **pipe behavior** overlays identify the conditions that give rise to starting and stopping activities in boxes, and that enable or disable the flow of data on pipes, respectively.

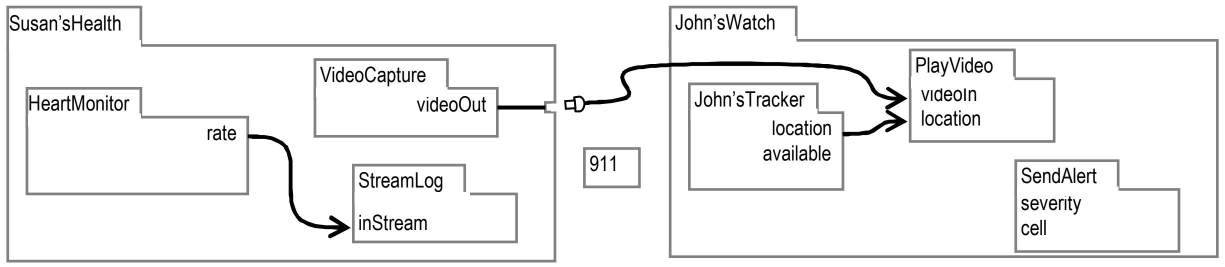
Conditions are expressions over the inputs and properties of the box they are associated with, or over the properties of the smaller boxes contained in the latter. In addition to operators such as equals (=), and (&), and or (|), conditions may include temporal operators such as *count*(c, t) that counts how many times condition c became true in the latest time interval t; or *sust*(c, t) which is true if condition c sustained a true value during the latest time interval t.

Wires transmit the result of evaluating a condition and may trigger one of three operations on boxes: start, pause, and stop, denoted by ▶, ❚❚ , and ■, respectively. Start operations may indicate the values of one or more inputs, which then should not be connected to pipes. The pause operation preserves the values of the properties and inputs to the box until a start is triggered again, possibly overriding some of those input values. A stop operation resets all the values in a box, being used, for instance, for privacy purposes.

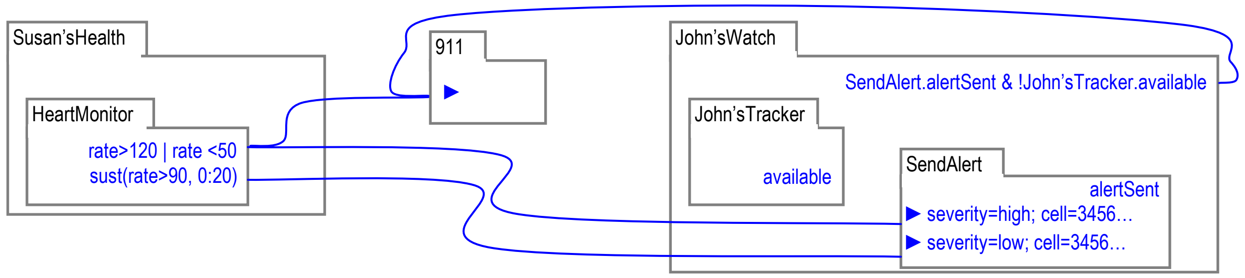
Valves can be placed on pipes, preventing the flow of data unless the enabling conditions are met. For example, the video output of a medical camera will not be released unless a potential emergency is declared.

4. Susan's heart condition

This section presents a case where an elderly lady, Susan, has developed a heart condition. Susan's doctor allowed her to return home, but wants her condition to



(a) Structure overlay



(b) Box behavior overlay

Figure 1. Monitoring Susan's heart

be constantly monitored.

For that, the doctor has created a box in uDesign for monitoring Susan's health, which wraps three services (see Figure 1(a), left hand side): heart rate monitoring, stream logging, for offline reference, and video capture. The latter is meant for checking on Susan remotely should a problem arise. The doctor also asked Susan if she would be interested in obtaining the devices to gather more sophisticated biometrics, such as skin galvanic response, but given Susan's current condition they agreed to leave those out for the moment.

The doctor used pipes to connect the monitored rate to the log input, and also to make the video output visible at the top level. After discussing Susan's lifestyle and physiological characteristics, the doctor identified two conditions to be monitored: when Susan's heart rate sustains a level above 90bpm (beats per minute) over 20 minutes, and when it either exceeds 120bpm or is short of 50bmp (Figure 1(b), left hand side).

To make it easy for Susan's family to recognize the prescribed conditions, the doctor names them *emergency* and *concern*. uDesign can show either these names or the expressions (as in the figure). The doctor also discusses the possibility of involving Susan's family as first-line responders to the conditions above, notwithstanding alerting emergency services.

Later at home, Susan discusses the doctor's prescription with her son John and they agree on alerting John if either condition is observed, and on alerting the emergency services in the event of an *emergency*, or if a *concern* condition arises but John is not available.

To coordinate the activities on his side, John defines the John'sWatch box where he includes services to follow his location and determine if he is available, and for alerting him over the cell phone network. The location service also helps determine the best device to map the PlayVideo service. John leaves the video pipe unhooked, to preserve Susan's privacy, planning to establish the connection only if the need arises. Alternatively, John could have used valves to control the flow of video on the pipe (see the next example).

5. Surveillance in John's home

John moved recently to a new house and made arrangements for a dog-sitter to come in during the day and walk his dog. However, John would like to be sure that the sitter does not venture into the private areas of the house. After work, John buys a couple of uDesign-enabled cameras and motion detectors. Upon powering up these devices at the home, uDesign's wireless discovery mechanisms pick them up, and John is able to assign them unique names within his house.

John deploys one camera and motion detector by the kitchen door, where the sitter will be coming in, and another camera and detector in the hallway leading

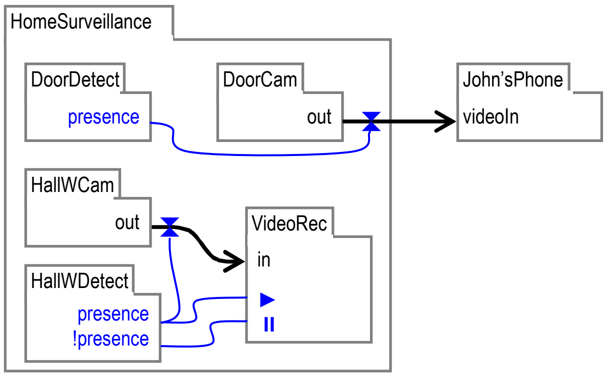


Figure 2. Surveillance in John's home

up to the main part of the house (Figure 2). Instead of installing a uDesign-enabled electric opener for the kitchen door, John just provided the sitter with a key.

To be aware of the sitter's movements, John uses uDesign to pipe the output of the door camera to his cell phone, places a valve on that pipe so that video only flows when someone is detected in the door area. John's cell phone will alert him of incoming video.

For the hallway camera, John chooses to record its output when a presence is detected, which John may review upon returning home. This is accomplished by placing a valve on the output of the hallway's camera, saving the home's wireless network from continuously piping video when no-one is in the hall.

6. uDesign Architectural Style

This section elaborates the uDesign architectural style using Zed [14], which defines both the architectural element types and the behavior and construction rules that comprise uDesign.

Figure 3 shows generic depictions of the elements, To reduce visual clutter, uDesign diagrams are organized in three overlays: structure, Figure 3(a), box behavior, Figure 3(b) top part, and pipe behavior, Figure 3(b) bottom part. To manage complexity, box decomposition is supported (Figure 3(c)).

Overlays are projections, or views, of the formal model of a system. When users edit a specification using overlays, consistency is guaranteed by an internal representation, the model of the system, which conforms to the semantics presented in this section.

The component types of uDesign are boxes, which have a name, inputs and properties; data stores, typically representing files. The connector type Pipe comes in two flavors: first, a producer-consumer kind, which connects box properties to inputs, and is represented as a single headed arrow in the direction of data flow. The second flavor supports read/write random access to data stores and is represented by double headed arrows (not further discussed here, for the sake of space).

Formally, the structure of a box is:

BoxStructure

$i, p: \mathbb{F}\ NAME;\ \ type: NAME \rightarrow\!\!\!\!\!\!\!+\ TYPE$

$i \cap p = \varnothing;\ \ \mathrm{dom}\ type = i \cup p$

where names in all caps are given sets (same throughout this section) and the constraints assert that inputs and properties must have distinct names, and also that all inputs and properties have a known type.

The behavior of a box is:

$Operation ::= start \mid pause \mid stop$

$Valuation == NAME \twoheadrightarrow\!\!\!\!\!+\ VALUE$

$Condition == \mathbb{F}\ NAME \times EXPR$

BoxBehavior

$op: ID \twoheadrightarrow\!\!\!\!\!+\ Operation$

$init: ID \twoheadrightarrow\!\!\!\!\!+\ Valuation$

$cond: ID \twoheadrightarrow\!\!\!\!\!+\ Condition$

$startName, condName: ID \twoheadrightarrow\!\!\!\!\!+\ NAME$

$\mathrm{dom}\ init \subseteq \mathrm{dom}\ (op \rhd \{start\})$

$\mathrm{dom}\ startName \subseteq \mathrm{dom}\ (op \rhd \{start\})$

$\mathrm{dom}\ cond \cap \mathrm{dom}\ op = \varnothing$

$\mathrm{dom}\ condName \subseteq \mathrm{dom}\ cond$

where operations are represented on the left hand side of a box (top of Figure 3(b)), and conditions to be monitored on the right. Start operations have an associated *valuation*, more below, which are enforced each time

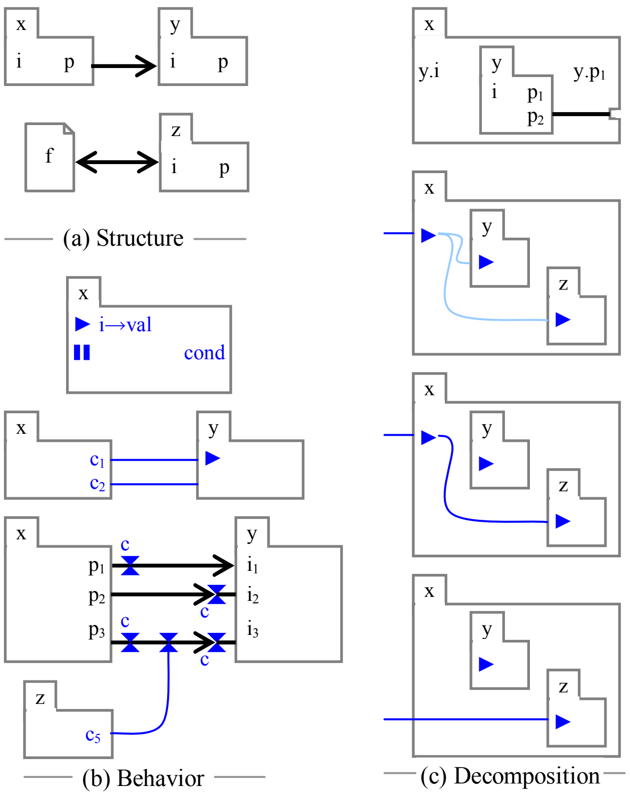


Figure 3. Syntactic primitives in uDesign

the corresponding start operation is activated.

Although operations (*op*) and conditions (*cond*) have an internal identifier (of type *ID*), start operations and conditions may also be associated with a name. This is to make them easier to recognize by non-expert users, who don't necessarily have deep domain knowledge. If a name is given, it will appear on the diagrams, although users may edit the corresponding expression on demand.

The *Box* schema coordinates structure and behavior:

Box

$name: GNAME$

$BoxStructure;\ BoxBehavior$

$\forall v: \mathrm{ran}\ init \bullet\ \mathrm{dom}\ v \subseteq i$

$\forall c: \mathrm{ran}\ cond;\ n: \mathbb{F}\ NAME;\ e: EXPR \bullet\ c = (n, e) \Rightarrow n \subseteq i \cup p$

Specifically, the domain of every valuation *v* is a subset of inputs, that is, a valuation maps some number of inputs to values; and every condition *c* is expressed as an expression written in terms of some number of inputs and properties of that same box.

There are two alternatives for naming boxes, which in the Zed model are both abstracted in the given set *GNAME*. First, a name may identify a specific entity or activity, such as Susan'sHealth or John'sKitchen. The structure of such names is *entity@environment*, where environments are uniquely identified and follow the usual conventions for URL "domains." Typically, environment names are assigned to independently administered geographic areas, such as homes and company buildings or campuses. For simplicity, the environment part of a name is omitted in diagrams for entities that reside in that environment. Data stores are also named in this fashion.

Second, a name may identify a generic entity of a given type, *name*:*type*, such as *s*:*Screen*. This indicates that the user is not concerned about identifying a specific screen to incorporate the system, but rather is willing to use one that is convenient. This corresponds to the notion of generic service. Resolving for a concrete service supplier is done dynamically, using optimized service discovery mechanisms such as the ones described in [12]. The naming of types follows the usual convention for defining ontology and name spaces in the internet, for which domain experts may provide local aliases to make names more recognizable to end-users.

$allOpenInputs: \mathbb{F}\ Box \rightarrow \mathbb{F}\ NAME$

$allProperties: \mathbb{F}\ Box \rightarrow \mathbb{F}\ NAME$

$\forall s: \mathbb{F}\ Box \bullet\ allOpenInputs\ s$
$\quad = \{\ n: NAME \mid \exists b: s \bullet \forall v: \mathrm{ran}\ b.\ init \bullet\ n \in b.\ i \wedge n \notin \mathrm{dom}\ v\ \}$

$\forall s: \mathbb{F}\ Box \bullet\ allProperties\ s = \{\ n: NAME \mid \exists b: s \bullet\ n \in b\ .\ p\ \}$

The definition of a system's structure uses generic operations *allOpenInputs* and *allProperties*, which extract the corresponding sets of names from a set of boxes. Open inputs are the ones that are not part of some valuation associated to a start operation.

A system's structure is characterized by a set of boxes, Figure 3(a), where no two distinct boxes have the same name, and by the pipes that interconnect box properties to open inputs of those boxes. The fact that pipes are formally defined as a (partial) function from inputs to properties means that a property may be piped to any number of inputs, but an input may receive data from at most one property.

SystemStructure

$$\begin{array}{|l}
\mathit{boxes}: \mathbb{F}\ \mathit{Box} \\
\mathit{pipes}: \mathit{NAME} ⇸ \mathit{NAME} \\
\hline
\forall b_1, b_2: \mathit{boxes} \bullet b_1 . \mathit{name} = b_2 . \mathit{name} \Rightarrow b_1 = b_2 \\
\mathrm{dom}\ \mathit{pipes} \subseteq \mathit{allOpenInputs\ boxes} \\
\mathrm{ran}\ \mathit{pipes} \subseteq \mathit{allProperties\ boxes}
\end{array}$$

Similarly, the definition of a system's behavior uses generic operations *allOps* and *allConds*, which extract the corresponding sets of ids from a set of boxes.

$$\begin{array}{|l}
\mathit{allOps}: \mathbb{F}\ \mathit{Box} \longrightarrow \mathbb{F}\ \mathit{ID} \\
\mathit{allConds}: \mathbb{F}\ \mathit{Box} \longrightarrow \mathbb{F}\ \mathit{ID} \\
\hline
\forall s: \mathbb{F}\ \mathit{Box} \bullet \mathit{allOps}\ s = \{\ \mathit{opId}: \mathit{ID} \mid \exists b: s \bullet \mathit{opId} \in \mathrm{dom}\ b . \mathit{op}\ \} \\
\forall s: \mathbb{F}\ \mathit{Box} \bullet \mathit{allConds}\ s = \{\ \mathit{cId}: \mathit{ID} \mid \exists b: s \bullet \mathit{cId} \in \mathrm{dom}\ b . \mathit{cond}\ \}
\end{array}$$

A system's behavior is characterized by the wires that connect conditions to operations, and by the valves placed on pipes. There are no restrictions to the wiring of conditions and operations: an operation is activated if any of the attached wires goes live, that is, if any of the related conditions hold.

SystemBehavior

$$\begin{array}{|l}
\mathit{boxes}: \mathbb{F}\ \mathit{Box};\quad \mathit{pipes}: \mathit{NAME} ⇸ \mathit{NAME} \\
\mathit{wires}: \mathit{ID} \leftrightarrow \mathit{ID} \\
\mathit{valves}: \mathit{ID} \leftrightarrow \mathit{NAME} \times \mathit{NAME} \\
\hline
\mathrm{dom}\ \mathit{wires} \subseteq \mathit{allConds\ boxes};\ \mathrm{ran}\ \mathit{wires} \subseteq \mathit{allOps\ boxes} \\
\mathrm{dom}\ \mathit{valves} \subseteq \mathit{allConds\ boxes};\ \forall p: \mathrm{ran}\ \mathit{valves} \bullet p \in \mathit{pipes}
\end{array}$$

Valves relate conditions with pipes. For data to flow in a pipe, it is necessary that all the valves are open, that is, that all the related conditions hold. The bottom of Figure 3(b) shows the diagrammatic representation of two special cases and the general case of valve representation. In the general case, illustrated by c_5 in the figure, a wire is shown connecting the condition and the commanded valve, ⧗. To reduce clutter, in case the pipe is attached to either an input or property of a box *b*, and the controlling condition also belongs to *b*, than the valve is shown next to the input (c_2 and c_4) or property (c_1 and c_3) and the wire is not shown.

A system coordinates structure and behavior:

$\mathit{System} \mathrel{\widehat{=}} \mathit{SystemStructure} \wedge \mathit{SystemBehavior}$

where *boxes* and *pipes* are shared among the two.

Operations such as adding a pipe or a valve to a system can also be modeled:

addPipe

$$\begin{array}{|l}
\Delta \mathit{SystemStructure} \\
\mathit{pipeSource?}: \mathit{NAME};\quad \mathit{pipeSink?}: \mathit{NAME} \\
\hline
\mathit{pipeSink?} \in \mathit{allOpenInputs\ boxes} \\
\mathit{pipeSource?} \in \mathit{allProperties\ boxes} \\
\mathit{boxes}' = \mathit{boxes} \\
\mathit{pipes}' = \mathit{pipes} \oplus \{(\mathit{pipeSink?} \mapsto \mathit{pipeSource?})\}
\end{array}$$

where the *pipeSource?* parameter must be an open input, and the *pipeSink?* a property in the system's set of boxes. As a result of adding a pipe, the *pipes* function in the system's structure includes the new mapping.

addValve

$$\begin{array}{|l}
\Delta \mathit{SystemBehavior};\quad \Xi \mathit{SystemStructure} \\
\mathit{cond?}: \mathit{ID};\quad \mathit{pipe?}: \mathit{NAME} \times \mathit{NAME} \\
\hline
\mathit{cond?} \in \mathit{allConds\ boxes};\quad \mathit{pipe?} \in \mathit{pipes} \\
\mathit{wires}' = \mathit{wires};\quad \mathit{valves}' = \mathit{valves} \oplus \{(\mathit{cond?} \mapsto \mathit{pipe?})\}
\end{array}$$

Adding a valve changes the system's behavior, but not its structure. As a result of the operation, the *valves* relation incorporates the mapping between the condition *cond?* and the *pipe?*, which must both be already defined in the system.

To manage complexity, a user may decide to wrap a system or part of a system as a box. Structurally, any property or input of the wrapped boxes may be promoted to the top level either using the dot notation, or by driving a pipe to the edge of the wrapping box, as illustrated at the top of Figure 3(c). Formally, the structure of a composite box combines the features of a *Box* with those of a *SystemStructure*:

CompositeStructure

$$\begin{array}{|l}
\mathit{Box};\quad \mathit{SystemStructure} \\
\mathit{iMap}: \mathit{NAME} ⤔ \mathit{NAME};\quad \mathit{pMap}: \mathit{NAME} ⤔ \mathit{NAME} \\
\hline
\mathrm{dom}\ \mathit{iMap} \subseteq \mathit{allOpenInputs\ boxes};\quad \mathrm{ran}\ \mathit{iMap} = i \\
\forall b: \mathit{boxes}; i_0, i_1: \mathit{NAME} \bullet \\
\qquad i_0 \in b . i \wedge \mathit{iMap}\ i_0 = i_1 \Rightarrow b . \mathit{type}\ i_0 = \mathit{type}\ i_1 \\
\mathrm{dom}\ \mathit{pMap} \subseteq \mathit{allProperties\ boxes};\quad \mathrm{ran}\ \mathit{pMap} = p \\
\forall b: \mathit{boxes}; p_0, p_1: \mathit{NAME} \bullet \\
\qquad p_0 \in b . p \wedge \mathit{pMap}\ p_0 = p_1 \Rightarrow b . \mathit{type}\ p_0 = \mathit{type}\ p_1
\end{array}$$

The partial surjections (one-to-one) *iMap* and *pMap* map some number of open inputs and properties of the wrapped boxes to the top-level box, such that the corresponding types are preserved. All the top-level inputs and properties are images in these mappings, that is, no inputs or properties can be defined at the top level that have no correspondence in the wrapped boxes.

The behavior of a composite box combines the features of a *Box* with those of a *SystemBehavior*:

CompositeBehavior

Box; SystemBehavior

$opMap: ID \leftrightarrow ID; \quad cMap: ID \rightarrowtail ID$

$\mathrm{dom}\ opMap \subseteq allOps\ boxes; \quad \mathrm{ran}\ opMap = \mathrm{dom}\ op$

$\mathrm{dom}\ cMap \subseteq allConds\ boxes; \quad \mathrm{ran}\ cMap \subseteq \mathrm{dom}\ cond$

Operations at both levels can be freely related. In general, one can imagine composites with dormant states, where a stop at the top level deactivates most wrapped components, while activating a few others. To cover common cases, the editing tools take as a default that an operation at the top level is mapped as the corresponding operation in all the wrapped boxes (lighter color wires in Figure 3(c)). That default can be overwritten by explicitly establishing the desired relation. Also, if a user has access privileges to see the internals of a composite, the editing tools support wiring directly to the wrapped boxes (bottom of Figure 3(c)) although semantically that corresponds to defining a condition/operation at top level which is then related to the wrapped box.

Additional conditions can be defined at the top level, using both the names of the conditions in the wrapped boxes as well as properties and inputs, using the dot notation as before.

Finally, the generic notion of *BOX* is either an elementary Box, or a composite:

$BOX \ \widehat{=}\ Box \vee CompositeStructure \wedge CompositeBehavior$

Recursive decomposition of systems would be modeled by changing the *SystemStructure* and *SystemBehavior* schemas above to refer of a set of *BOX*, rather than *Box*.

7. uDesign implementation

This section describes how uDesign maps to activity-oriented systems (e.g. [13]), and to a specific implementation thereof: the Aura infrastructure. We start by summarizing Aura (details can be found in [11].)

Activities, or tasks, are a first class concept in Aura and correspond to everyday activities of users or automated agents; for example, *write paper X*, or *monitor Susan's health*. Activity models describe the services and materials required to support the activity, as well as user preferences concerning the selection and configuration of those services.

Services correspond to features such as *edit text*, or *monitor heart rate*, or to capabilities of humans, such as *answer the door*. Materials correspond to data stores manipulated by services. Services may define ports

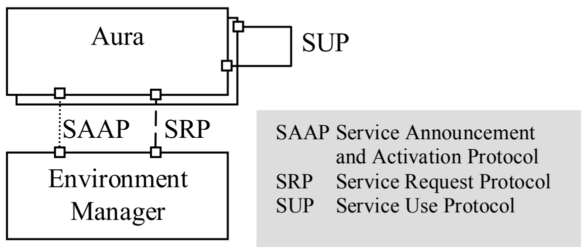

Aura
SUP
SAAP
SRP
Environment Manager
SAAP Service Announcement and Activation Protocol
SRP Service Request Protocol
SUP Service Use Protocol

Figure 4. The Aura Architecture

[10], and activity models may then include service interconnections via typed connectors.

The Aura infrastructure takes activity models, represented in XML, finds appropriate service suppliers in the environment and, upon request, assembles and activates those services.

Figure 4 shows the architecture of the Aura infrastructure, with two component types, *Aura* and the *Environment Manager* (EM), and the three interaction protocols (connectors) between them. The double box for the Aura type indicates that typically there will be many Aura component instances in each environment, while the EM (single box) will have only one instance.

The EM provides the mechanisms to optimally marshal the supply of services required by activities [12]. It also monitors these services to ensure that they are satisfying user goals, recommending reconfigurations where appropriate. Environments typically correspond to a physical space (e.g., a floor or a building), but they are defined administratively and so can also encompass logical spaces (e.g., including a printer at the office in the home environment).

While an EM contains generic mechanisms for service discovery and configuration, Auras contain domain-specific knowledge about activities and services.

Auras are abstract models of entities in the real world, such as users, spaces, devices, and software components. Auras provide mechanisms to monitor and control the entities they represent. In the case of software, Auras may wrap existing applications to conform to the infrastructure's protocols. Rather than requiring writing a new portfolio of applications, this approach makes it easy to integrate legacy applications into the Aura infrastructure. For instance Emacs, MSWord and Notepad have been wrapped to become Auras that offer *edit text* services. Such Auras act as translators between the generic configuration directives issued by the EM and other Auras, and the specific APIs offered by the component they encapsulate.

Earlier versions of the architecture separated the roles of service supplier and consumer into two component types: Supplier and Prism [11]. However, the work leading up to [13] made us realize that the same component, an Aura, can play either the role of a task manager (Prism) or of a Supplier of services, or both. For example, John's Aura manages John's activities,

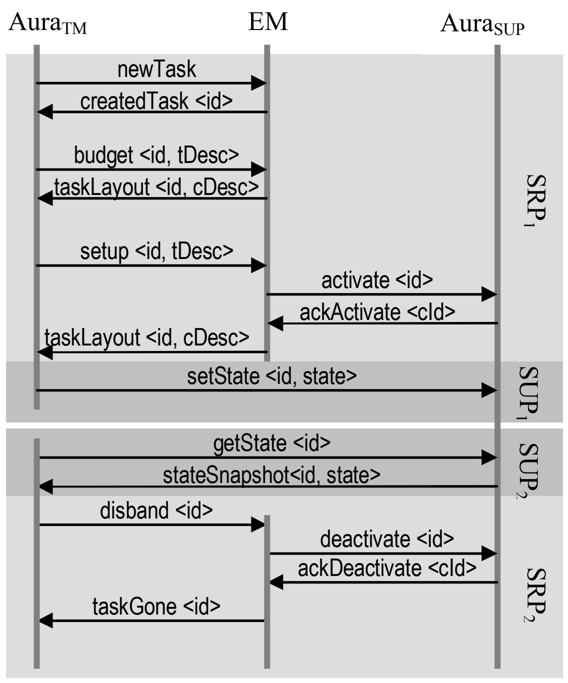


Figure 5. Task lifecycle in Aura

such as looking after Susan, but also offers services that John can provide, such as answering the door.

An Aura that plays the role of a service supplier, denoted Aura$_{SUP}$, announces its services with the EM via the Service Announcement and Activation Protocol (SAAP) connector, and finds itself on the supplying end of the Service Use Protocol (SUP) connector (see Figure 4). An Aura that plays the role of a task manager, denoted Aura$_{TM}$, requests the services required for each task from the EM via the Service Request Protocol (SRP) connector, and finds itself on the consuming end of the SUP connector.

When a user enters an environment, an Aura$_{TM}$ is associated with him or her to manage the user's tasks. A task model contains information about the desired services that should be brought to bear to help the user carry out the task. To resume a task in a particular environment, Aura$_{TM}$ communicates with EM to request service suppliers using a combination of the protocols. These protocols, shown in Figure 5, starts with Aura$_{TM}$ opening a session for a particular task with newTask; after getting a reply from EM, a unique session id is used for subsequent communication about this task. Aura$_{TM}$ then obtains estimates for the how well the environment can support a particular task by sending a budget message and getting taskLayout messages in response. It is possible that more than one way of instantiating a task depending on the degree of richness of service suppliers in the environment (e.g., requesting an editText service might be provided by a NotePad ssupplier or a Word supplier). Aura$_{TM}$ learns user preferences which guide the choice of service suppliers. The EM will then activate those Auras$_{SUP}$.

Once an Aura$_{SUP}$ has been activated by the EM, Aura$_{TM}$ interacts with it to set its state so that the user can start using them. A state might include the materials (files) to use, the size of windows, cursor positions, etc. When the user is finished with the task in that environment, Aura$_{TM}$ will issue a getState on each of the service suppliers to get the updated information about their state, and then will request that EM disband the task. This deactivates each of the Auras$_{SUP}$ and closes the session between the EM and Aura$_{TM}$ for that task.

7.1 Mapping uDesign to Aura

uDesign takes a unified view of Auras, activities, and services. Auras that just offer services correspond to boxes with no internal structure. Auras that manage activities (Aura$_{TM}$) correspond to composite boxes, and the activities themselves correspond to subsystems inside the composite box. Resuming an activity in Aura corresponds to a start operation in a composite box that activates the subsystem corresponding to the activity.

The structure overlay in uDesign corresponds to defining which services and materials are required to support an activity. For example, in Aura, Susan's-Health is one of the activities of user Susan, and it requires services HeartMonitor, VideoCapture and so forth. In uDesign, Susan'sHealth is a composite box that wraps boxes HeartMonitor, VideoCapture and so forth (see Figure 1).

Inputs and properties of boxes map to service ports in Aura, which can then be interconnected by connectors of type pipe.

The behavior overlay in uDesign corresponds to defining under which conditions activities/services are to be activated. The way current Aura protocols support uDesign behavior is better illustrated with an example: the Aura$_{TM}$ for HomeSurveillance issues a setState instructing the Aura$_{SUP}$ for HallWDetect to monitor the *presence* condition, as defined in Figure 2. When the condition holds, the Aura$_{SUP}$ issues a stateSnaphot describing the event. The Aura$_{TM}$ then uses the SRP and SUP to start an Aura$_{SUP}$ for video recording.

Although this implementation strategy works to coordinate services and activities within the same Aura, it becomes cumbersome for achieving coordination across different Auras. Since uDesign allows end-users to extend wires across systems, we are currently designing extensions to the Aura protocols to support

Table 1. uDesign behavior vs. Aura protocols

operation	Aura protocol activity
▶	SRP$_1$ → SUP$_1$ (Resume)
❚❚	SUP$_2$ → SRP$_2$ (Suspend)
■	SRP$_2$ (Stop)

direct coordination between Auras.

Table 1 summarizes how the three operations in uDesign are realized using the protocols in the Aura architecture. Start, ►, corresponds to resuming an activity: SRP_1 followed by SUP_1 in Figure 5. The distinction between pause, ǁ, and stop, ■, is that the latter does not capture the current state of the suppliers.

Finally, valves enable the flow of data in uDesign pipes. Depending on the implementation of pipe connectors, such enabling conditions may be communicated to the pipe, or they may have to be communicated to the sending port.

8. Conclusion and future work

In previous work, like many other researchers, we have focused on building a software infrastructure over which enhanced applications for pervasive computing could be built and deployed. Specifically, we built the Aura infrastructure, which promotes user activities to first class primitives in software systems.

However, the means for end users to assemble and configure highly personalized and flexible solutions have been largely missing. Although infrastructures such as Aura and others make it easier to develop solutions for smart spaces, the mechanisms for interconnecting and coordinating components have remained at a level of detail more appropriate to computer specialists. This is precisely the gap addressed by uDesign.

Our main goal is to provide a method of connecting services that is appealing to a large user base by making it similar to connecting consumer electronics. Similar composition approaches are beginning to dominate domains such as business environments and robotics.

uDesign offers overlays to capture the structure and behavior of a system. Such separation is a recognized good practice in design methodology, and the formal basis (described in Section 6) of uDesign guarantees that no inconsistencies will arise from having separate views. Checking the system structure follows the formal model.

uDesign leverages concepts of software architecture and this paper shows it can be implemented on top of the existing Aura infrastructure with minor extensions. This work also clarifies the APIs and architectural assumptions that individual components must support to be integrated into the proposed framework.

Part of our working hypothesis is that uDesign will be accessible to a broad range of users, including non-experts. This hypothesis needs further validation. Specifically, we plan to conduct user studies involving both end users and domain specialists in domains such as assisted living and home security. This will involve developing a set of tools that are robust and usable by end-users in a real setting, rather than a lab setting. In these studies we will also assess the scalability of our approach. We believe that the hierarchical decomposition of boxes will aid in addressing this, and we also do not anticipate that activities will be excessively complex to warrant other specialized mechanisms.

References

[1] N. Georgantas, S. Mokhtar Y.-D. Bromberg, Yerom, V. Issarny, J. Kalaoja, J. Kantarovitch, A. Gerodolle, R, Mevissen, "The Amigo Service Architecture for the Open Networked Home Environment". Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05), pp. 295-296, 2005.

[2] J.S. Gero. Categorizing Technological Knowledge From a Design Methodological Perspective. Conference 'Technological Knowledge: Philosophical Reflections', Boxmeer, The Netherlands, 2002.

[3] GraphLogic Inc. PointDragon. http://pointdragon.com.

[4] I. Han; H.-S. Park; Y.-K. Jeong; K.-R. Park. An integrated home server for communication, broadcast reception, and home automation, Consumer Electronics, IEEE Transactions on, Vol 52(1), 2006.

[5] Microsoft Inc. Microsoft Robotics Studio Developer Center. http://msdn2.microsoft.com/en-us/robotics/default.aspx, accessed September 2007.

[6] U. Norbisrath, I. Armac, D. Retkowitz, P. Salumaa: Modeling eHome Systems. S. Terzis (ed.): 4th Intl Workshop on Middleware for Pervasive and Ad-Hoc Computing, Melbourne, Australia. ACM Press, 2006.

[7] U. Norbisrath, C. Mosler, I. Armac: The eHome Configurator Tool Suite. 1st Intl Workshop on Pervasive Systems (PerSys 2006), Montpellier, France, 30-31 2006, LNCS 4278, p. 1315-1324, Springer, 2006.

[8] C. Ouyanga, E. Verbeekb, W. van der Aalsta, S. Breutela, M. Dumasa, A ter Hofstedea. Formal semantics and analysis of control flow in WS-BPEL. *Science of Computer Programming* Volume 67, Issues 2-3, 1 July 2007, Pages 162-198.

[9] N.W. Paton (Ed.). Active Rules in Database Systems. *Monographs in Computer Science,* Spring, 1998.

[10] M. Shaw, D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[11] J.P. Sousa, D. Garlan. The Aura Software Architecture: an Infrastructure for Ubiquitous Computing. Carnegie Mellon Technical Report, CMU-CS-03-183, 2003.

[12] J.P. Sousa, V. Poladian, D. Garlan, B. Schmerl, M. Shaw. Task-Based Adaptation for Ubiquitous Computing. In *IEEE Trans on Systems, Man, and Cybernetics, Part C: Applications and Reviews, Special Issue on Engineering Autonomic Systems*, Vol. 36(3), May 2006

[13] J.P. Sousa, B. Schmerl, P. Steenkiste and D. Garlan. Activity-oriented Computing. In S. Mostefaoui, Z. Maamar and G. Giaglis (Eds), *Advances in Ubiquitous Computing: Future Paradigms and Directions*, IGI Publishing, Herschey, PA, 2008

[14] The Z Notation: A Reference Manual. Prentice Hall Intl Series in Computer Science, Prentice-Hall, 1992.

[15] S. White, Using BPMN to Model a BPEL Process - all 4 versions »BPTrends, 2005. http://businessprocesstrends.com.