Towards Multi-Design of Situated Service-Oriented Systems

João Pedro Sousa, Zeynep Zengin, Sam Malek
Computer Science Department, George Mason University
4400 University Drive, 4A5, Fairfax VA 22030, USA
ph: +1-703 993 1530
{jpsousa, zzengin, smalek}@gmu.edu

ABSTRACT

This paper discusses ongoing changes to the boundaries and roles of design and run time in the software lifecycle. Specifically, it focuses on changes caused by the emergence of situated systems in open pervasive computing environments. Clearly, such changes have a direct repercussion on the roles and tasks of system developers, stakeholders, and users.

The paper proposes extensions to current software design notations, concerning (a) service discovery and ways to scope it to user-defined physical locations, and (b) the ability to incorporate and shed features and behaviors at run time, depending on which users are present and on their goals, and including the ability to resolve conflicts between such goals. Five small but illustrative example systems demonstrate the benefits of these extensions.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design – *representations*.

General Terms

Design.

Keywords

Pervasive computing, user-centric design, situated systems, multi-user, conflict resolution.

1. INTRODUCTION

The boundaries of design and run time in the software lifecycle are in a state of flux. The push for self-healing and self-adaptive systems propelled *service discovery at runtime*: the decision of which component to invoke used to be made during design in component-based software and early service-oriented systems but can now be made automatically at run time [1,2].

Paradoxically, that shift of responsibility from design to run time created the need to expand design, enriching it with specifications of service types and constraints on quality of service (QoS). It also created the need for machine-readable models to guide automatic service discovery at run time: service ontologies and registries, models to estimate delivered QoS, etc.

A further push to optimize QoS led to *self-architecting*: the decision of which architectural pattern to use to promote a certain aspect of quality for a given feature is no longer necessarily made during design and can now be made automatically during deployment and re-examined as needed at run time [3,4]. This other shift of responsibility from design to run time created the need for machine-readable run-time models of architectural patterns and of the QoS aspects they promote or detract.

This paper focuses on an additional shift of responsibilities that results from the push towards pervasive computing. Pervasive computing goes beyond the use of mobile devices, encompassing the use of devices embedded or scattered in spaces ranging from homes to subway stations to outdoors such as streets and forests.

The merge of computation and the physical world requires systems to, first, become aware of the spatial relations between users and parts of the system, and second, to serve the multiple users who may be present in a space. For that, run-time service discovery must additionally become *situated*, and self-architecting must take into account the goals of *multiple users*.

Our work takes a user-centric perspective of multi-user situated systems. A traditional multi-user system is endowed with a set of features and a design that enables it to serve multiple users. In contrast, the kinds of systems we target are capable of automatically adopting *multiple designs* and adjusting their features to better serve *each* user who approaches the system with individual goals and expectations in mind.

As before, these added responsibilities at run time entail extending design specifications, e.g., with rules for recognizing situations and deciding when to adopt a given design. Such rules must be entirely machine readable and carry enough semantics to enable the automatic deployment and adjustment of designs.

The contributions of this paper include proposed extensions to a design notation towards (a) *situating* service discovery, and (b) specifying *multi-design*, i.e. the ability to serve multiple user-specific goals, including resolution should the goals of the users who are present at a specific time be conflicting in some way. The baseline notation is SAS (Service Activity Schemas), which is based on OMG's BPMN [5], and was introduced in prior work on self-architecting: SASSY [4,6].

Clearly, the profound changes in the roles of design and run time described above affect the roles of software engineers, stakeholders, and users in the software lifecycle.

In the remainder of this paper, Section 2 elaborates on the challenges, and contrasts the work presented here with related work. Section 3 summarizes the SAS notation and the relevant concepts of self-architecting. Sections 4 and 5 present the proposed SAS extensions by example, respectively for situated discovery and for multi-design. Section 6 discusses limitations and Section 7 summarizes the main points of the paper.

2. CHALLENGES AND RELATED WORK

Situated systems are an exciting new area with applications to managing heating and cooling in buildings, surveillance, assisted living and healthcare, transportation, emergency response, etc.

A body of work originates in mobile computing and explores applications such as context aware reminders and navigation, medicine cabinets, etc. [7,8]. Here, researchers have focused on

PESOS '10, May 1-2, 2010, Cape Town, South Africa
Copyright 2010 ACM 978-1-60558-963-3/10/05 ...$10.00.

building prototypes pushing the limits of technology and evaluating user acceptance and engagement. From a software engineering point of view, these are embedded custom-made applications where decisions are captured at code level.

Larger applications have been built for spaces such as hospitals, where the components, their interconnections, and permissible flows of information are carefully crafted and controlled [9,10].

It is harder to develop situated systems for open environments such as homes, office buildings, or mass transportation. Although existing situated systems demonstrate exciting new capabilities, it is unclear how to achieve scalability and adaptability to situations where users and devices come and go freely. Such applications may easily overstep the users' boundaries, causing users to feel at the mercy, rather than in control of technology [11]. For example, suppose that a user programmed a clothes dryer to automatically interrupt its activity during energy peak hours, but somebody else who doesn't share the same concern wants to use the dryer. Logic that might work in the context of a personal application running on a cell phone becomes rigid and brittle in shared spaces.

The goal becomes applying good practices of software design and construction to situated systems, capturing important decisions at the design level, and using proven approaches to achieve openness and adaptability, such as service orientation.

Activity languages such as BPMN raise the level of abstraction of design. They neither represent directly the code view nor the run time view of a system, but they capture *what* the system is intended to achieve. Recent work automatically translates BPMN schemas into views of service-oriented systems, e.g. [12].

SAS's precise semantics enables taking the next step and fully generate and deploy systems automatically. In SASSY, the equivalents of the traditional code and run-time views of a system become internal representations generated by tools and manipulated by run-time infrastructures [4]. Like BPMN, SAS is intended for use by domain experts.

BPMN targets primarily computational activities that compute some result and terminate. Services are frequently invoked synchronously and are expected to return a result as soon as possible. In addition to computational activities, SAS activities may last for a long time, supporting human activities or carrying out tasks autonomously for years. Services are frequently provided by devices, such as smoke detectors, which run asynchronously once started and communicate when necessary.

Activity-oriented computing (AoC) targets supporting lasting human activities in pervasive computing environments [13,14]. AoC shares with SASSY the principle that individual users may define their own activities, which may then be deployed, interrupted, and resumed at will and at run time by leveraging a service-oriented infrastructure.

Existing work in AoC differs from SASSY in two ways. First, it defines no activity design language. Models of activities are captured implicitly and consist of describing a collection of applications and files that support an end user in activities such as analyzing the results of medical exams and editing documents. Second, there is no provision for users to scope service discovery, which is implicitly done within the network domain the user is in.

More broadly, different kinds of service discovery are in use today. A significant number of systems are currently developed using mechanisms such as the Web Services Description Language (WSDL [15]), or Universal Description, Discovery and Integration (UDDI [16]). These mechanisms support service descriptions meant for human eyes to inspect during system development, and the decision of which service provider to invoke is then set in the code.

A variety of other mechanisms enable service discovery at run time, such as Microsoft's UPnP, IETF's SLP, etc. [2]. Ontology plays a key role in matching requests from service consumers with announcements from service providers. However, mechanisms to guide discovery at run time according to QoS goals are seldom available, and commonly, the decision of how to scope discovery geographically is set by some combination of coding and administrative network configuration.

To the authors' best knowledge the work herein is the first to promote situated discovery to a first-class construct at the design level. SAS additionally supports QoS-guided discovery at run time by associating utility functions with scenarios [4].

Multi-design is distinct from self-modifying software. A self-modifying program may alter some of its own code, either as a result of initialization parameters or upon reaching a certain state, usually to improve performance [17]. In contrast, activity schemas are not self-modifiable, but they contain rules for modifying the run-time organization of the system, i.e. for changing which services are marshaled and how they are interconnected.

Such rules trigger changes to the system design upon observing changes in situation, such as user location or the presence of other users. Specifically, when multiple users share a system or parts thereof, conflicts may arise as to the intended behaviors, setting of properties such as ambient temperature, etc.

Several communities have contributed bodies of work in conflict resolution. In economics, *auctions* support a form of negotiation among multiple parties competing for a limited resource [18]. The agents community developed protocols for multi-attribute *negotiation* [19], and *mechanism design* is a subfield of game theory that investigates incentive-punishment mechanisms for promoting desired behaviors among independent actors [20].

To the authors' best knowledge the work herein is the first to promote resolution mechanisms to the design level. This enables stakeholders to design their own solutions for resolution, choosing among a variety of mechanisms, and changing that choice as needed. This is in contrast with other work where a mechanism that promotes a specific model of resolution is chosen and imparted in the code. For example, current work in context-aware smart spaces, frequently uses some form of priority schema to automatically resolve conflicts among users [21].

3. SASSY AND EXTENSIONS

SAS is a graphical language for modeling activities in terms of functional, coordination, and QoS requirements for service-oriented systems. A complete description of the baseline SAS, a comparison with related languages, and description of tool support can be found in [6]. Here we summarize the SAS constructs used in this paper and discuss the relation between activity schemas and running systems.

Figure 1 summarizes SAS. Plain round-corner rectangles refer either to sub-activities, for hierarchical decomposition of schemas, or to system-supported activities, e.g., delay used for timing.

Gateways follow the notation used in BPMN: a diamond labeled with a **+** for parallel gateways, and with an **o**, for inclusive gateways. Parallel gateways relay a message to each outgoing arc once all incoming arcs have received a message. Inclusive gateways do so as soon as a message arrives in one incoming arc. Gateways may marshal incoming message parameters to outgoing message parameters as needed. Also, sending a message on an

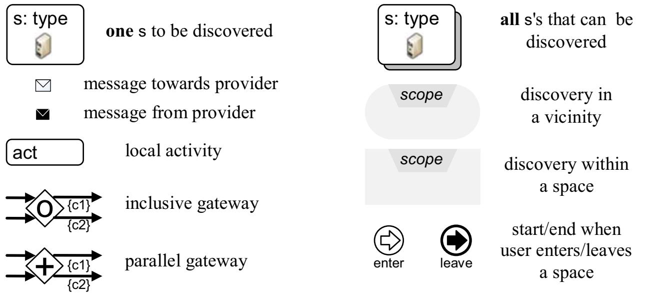


Figure 1. Baseline syntax of SAS, on the left, and proposed extensions, on the right.

outgoing arc may be further constrained by a condition associated to the arc, e.g., expressed as a function of message parameters.

Round corner rectangles with a server icon denote services to be discovered: a single box for one service, a stacked box for all services of the indicated type that can be discovered. Messages to and from service providers are indicated by white and black envelopes, respectively. When an envelope is next to a set of services (stacked box), the notation provider.message is used to specify the source/target provider. Message parameters may be shown in parenthesis, when relevant, but the SAS tools support editing the definition of a message as needed.

In the proposed extensions, spaces are defined administratively and used as units to manage trust and authorization for accessing the services within each space. Some spaces may be physically occupied by users, such as rooms and cars, and may be contained in other spaces, e.g., a room within a building. Spaces may be confined to a device, such as a cell phone, which at times acts as a user's only computational support. The constructs shown in the figure are explained by example in Section 4.

The set of activity schemas defined and available to be initiated in a space changes as users and their devices come and go. Users initiate and retire activities at will. For starting an activity, SASSY discovers the required services and deploys a service coordinator thread which reads the activity schema and performs the desired coordination logic.

While some activities run to completion (computational activities typically do), lasting activities such as controlling the temperature in a room may last until situational events are observed (see example in Figure 6) or the user decides to retire the activity (see example in Figure 3).

Figure 2 informally depicts an example with two spaces: a house and an office building. Two of the currently available activity schemas at the house have been deployed, one of which marshaled a service at the office (see also Figure 4).

Traditional notions of software evolution and adaptation fail to capture the highly dynamic relation between activity schemas and running systems in open, pervasive computing spaces. The design of a situated system, i.e., which components it includes and how they are interconnected, changes with the activities that multiple users bring to the space and decide to initiate, thus adding features and behaviors to the running system. We call this *multi-design*.

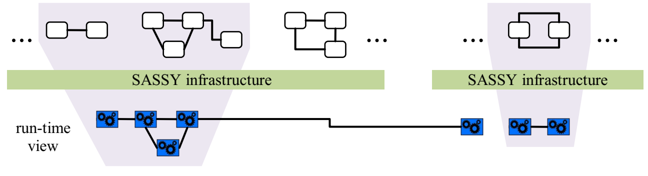


Figure 2. Relation between activity schemas and systems.

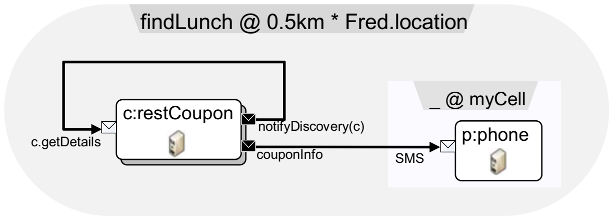


Figure 3. Example of vicinity discovery: Fred receives an SMS for each restaurant coupon found within 500 meters of Fred's location, possibly while he walks around the city.

4. SITUATED DISCOVERY

In situated systems, service discovery must itself become situated. Because the components of a situated system interact with their physical surroundings, it matters where those components are deployed: traditional situated systems are often embedded in a target device, such as a phone or a medicine cabinet [8,22]. This is in contrast with web-based systems, in which services may run anywhere on the network as long as functional and quality requirements are met.

The tenets of situated discovery are that: (a) situated services are aware of, and announce their location, and (b) service requests are scoped to a geographic area. We distinguish two kinds of scoping for situated discovery: one in the vicinity of a mobile entity, such as a person or device; the other within fixed spaces, such as rooms, buildings, and streets.

Figure 3 shows an example activity schema where discovery is scoped to the vicinity of a user. All services shown on top of the rounded shaded area are to be discovered within a radius of the location indicated in the area's label, which is of the form: name @ radius * location, to be read, name *at* radius *around* location.

Sensing the location of users may be implemented by a variety of mechanisms based either on following proxy devices such as cell phones, or on mechanisms embedded on spaces such as face recognition and id card readers, and which may be combined for better accuracy [23]. At the design level, the location of entities known to a system is referred to in the form entityId.*location*.

The stacked service boxes for c:restCoupon in the figure indicates that *all* service instances of type restCoupon within the specified vicinity are to be marshaled into the system. As with regular discovery, this could be guided by additional constraints on properties and quality of the services.

Because the discovery of such instances may occur at any time while Fred moves around, the design captures a notification issued by the *discovery infrastructure* (not by the provider) for each service that comes in range: notifyDiscovery. The notification includes the identity of the discovered service, which can be used to direct subsequent calls, as in the example.

This example illustrates how activity schemas can situate the discovery of services in the vicinity of a mobile user, and also capture at the design level how to react to the dynamic discovery of new services that come into range.

To illustrate scoping discovery to spaces, consider the example of an alarm system at Fred's home. When motion detectors are triggered, Fred receives a phone call at the location he currently is, which may be directed to his cell phone if he happens to be carrying it. Fred may press keys on the phone to interact with the system; for example, to ask video to be streamed to a screen in that same space, or to indicate a false alarm. Video capture follows the possible intruder by activating a camera in the vicinity of the latest motion detector to be triggered. If Fred cannot be

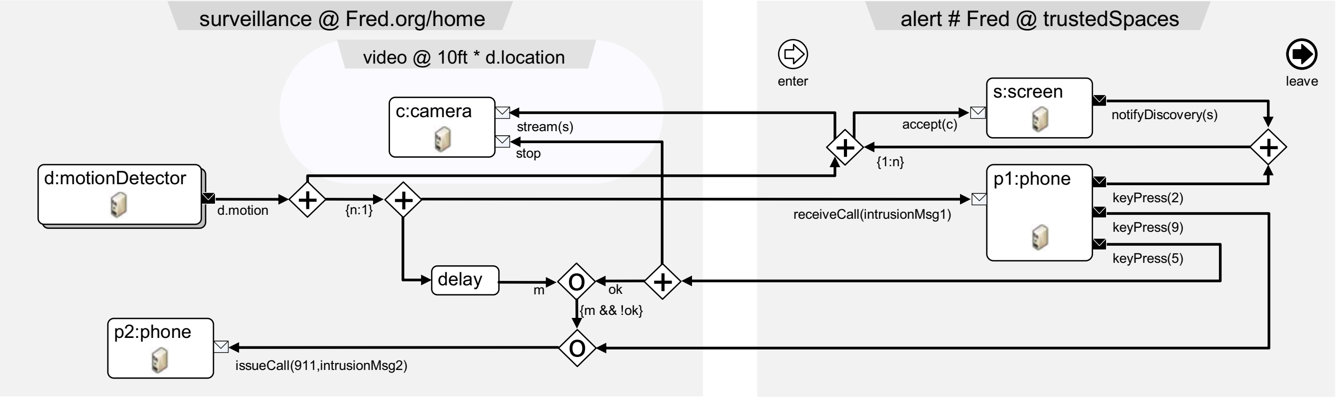


Figure 4. Fred's home surveillance system is a distributed system that includes services at Fred's home and at whichever location Fred happens to be, among a defined set of trusted locations. Fred entering and leaving one such location has no computational effect other than establishing the scope of service discovery for alerting purposes.

reached within a set time, a call to 911 is automatically placed.

Figure 4 shows the corresponding activity schema. All services on top of a shaded rectangle are to be discovered within the space indicated on the rectangle's label. The rectangle on the left refers to the part of the system deployed at Fred's home. Fixed locations are named using a notation with support for aliasing similar to URLs on the internet. For example, fairfax.va.us/22030/university-drive/4400 represents the same space as gmu.edu.

The rectangle on the right refers to the space Fred happens to be in. This is indicated by a label of the form name # entity @ possibleSpace, to be read, name *for* entity *at* possibleLocation. The entity's location is tracked as for vicinity discovery and matched for containment in possibleSpace. When the entity enters a matching space an enter event is issued, and conversely, upon leaving the space, a leave event is issued.

In this example, Fred.location is tracked and matched against the alias trustedSpaces, which refers to a list of spaces defined by Fred, such as his office, or a virtual "space" defined by a device such as his laptop or cell phone. The most likely space, physical or virtual, Fred is actually in is judged by the location mechanisms for determining Fred.location [23].

As before, the stacked boxes for d:motionDetector refer to all services of that type discovered within Fred's home. Whenever one of those services, d, issues a motion detected message, d.location is used to scope the discovery of a camera within 10 ft.

The {n:1} label on the incoming arc to the parallel gateway prevents each subsequent d.motion from initiating an extra phone call. Also, the {1:n} label associated with the screen discovery and Fred's consent enables the redirection of video streaming from new cameras following each activated detector.

This example illustrates the discovery of services within spaces, both at a fixed location and at a location identified by the presence of an entity of interest, such as a user. The example introduces the enter and leave events, which signal the presence of entities in spaces, and here these events are used to enable the part of the system for alerting Fred. As in the previous example, this one also illustrates vicinity discovery, this time following the location of varying entities: the newly triggered motion detectors.

5. MULTI-DESIGN

The fundamental tenet of multi-design is that the design of a situated system depends on which users are present, and on the intent of those users. In other words, the function and design of a system may change at run time, as multiple users come and go bringing their intentions and expectations. This is in contrast with traditional software systems, where there is a clear demarcation between design and run time, and often a complex, human labor-intense process to deploy a system.

Different users think and act independently and frequently approach a situated system with different intentions and preferences. For example, when sharing a car, some users may prefer to enjoy the landscape while others prefer to minimize the duration of the trip. As another example, when two users exchange a large volume of data in real time, one user may prefer to make communication secure while the other may be in a rush and unwilling to bear the overhead of encryption.

Differences such as these need to be resolved during the self-configuration of a situated system serving multiple users. In the remainder of this section, we discuss different kinds of resolution.

5.1 Resolving Features

Each user may expect a different set of features or activities to be automatically carried out by a situated system. For example, suppose that Bob installed a smart power meter at his home which issues pricing signals reflecting the load on the power grid [24]. Because the clothes dryer is a heavy energy consumer, Bob would like it to suspend drying during high rates and to resume only when night rates are in effect. However, Bob's wife Mary would like to have her clothes dry regardless of such issues.

Figure 5 shows the activity schema for saving energy defined by Bob. The activity starts when Bob enters the laundry room, and then the dryer receives suspend and resume messages corresponding to the rates announced by the smart meter. The activity finishes once the drying cycle is done or if Bob leaves the laundry room without having started the dryer.

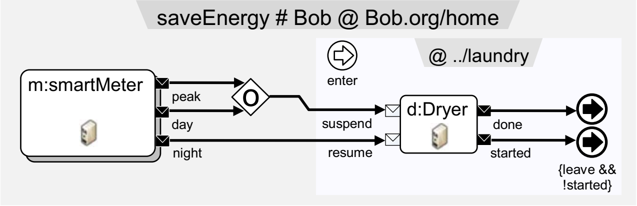
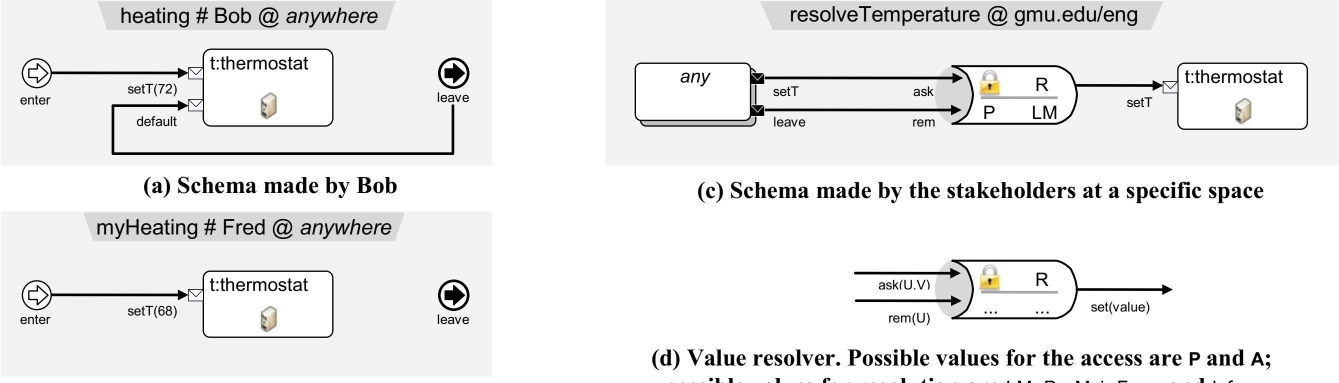


Figure 5. Saving energy at Bob's house: an example of activating a system feature depending on who is present.



(a) Schema made by Bob

(b) Schema made by Fred

(c) Schema made by the stakeholders at a specific space

(d) Value resolver. Possible values for the access are P and A; possible values for resolution are LM, Pr, Maj, Freq, and Infreq

Figure 6. Example of how different designs concerning the usage of a shared physical service can be reconciled. In this example, reconciliation concerns the value to be set on the service.

This activity, and the interconnections between the smart meter and the dryer that support it, only take place when it is Bob who is present in the laundry room. Should Mary agree to have this feature activated for her as well, Bob could add Mary to the label of the schema: saveEnergy # Bob, Mary @ etc., or Mary could have her own activity schema reflecting her preferences; for example, interrupting the dryer only when peak rates are in effect.

The activity schema in Figure 5 clearly reflects at the design level the relationship between a user, a situation, and an expected feature in that situation. Such activity schemas are used by a situated system to determine the required design, i.e. which services to marshal and how to interconnect those services, depending on which users are present.

5.2 Resolving Values

An important kind of services is characterized by a value, or values desired by users. For example, the channel on a radio or TV, the sound volume on the same, the level of lighting in a room, or the temperature set for heating and air conditioning.

Multiple users may intentionally share a service of that kind, or sharing may be incidental by virtue of being present at the same space. In such cases, the situated system can do a better job at serving the users, first, if their preferences are known, and second, if the differences among such preferences can be resolved in a way perceived as fair by the users.

The ability to automatically take into account the preferences of individual users present at a space represents a significant improvement over current practice. Today, building automation systems adopt one-size-fits-all settings, typically based on a standard work schedule and oblivious of which spaces are actually occupied, let alone of the preferences of different occupants.

Figure 6 shows an example concerning the temperature setting for heating systems in spaces such as rooms and vehicles. Users express their preferences by creating activity schemas that discover the thermostat for the space and, upon detecting the user entering the space, set the thermostat for the preferred temperature. Parts (a) and (b) show examples of such schemas created by Bob, who prefers 72º Fahrenheit, and by Fred, who prefers 68. Both users made their schemas unconstrained as far as location by labeling the location with the keyword *anywhere*.

Bob's schema reflects his concern to reset the thermostat back to the default temperature once he leaves the space, while Fred was not sure what to do, since there might be other people left in the room, so he did not specify an action for when leaving.

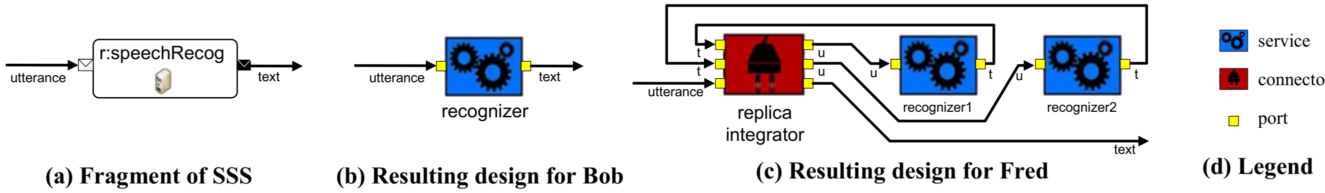
Part (c) of the figure shows the activity schema created by the stakeholders at a specific space, the eng building at gmu.edu in this example, for regulating the access to the thermostat service. Here, the communication between *any* activity and the thermostat goes through a *value resolver* that enforces a least misery (LM) policy. This resolver outputs the average of the current requests, or a default value set by the stakeholders when no requests are outstanding. This activity schema accommodates any activity that issues setT messages to a thermostat and that detects and is willing to communicate when the user leaves the space.

The resolution process supported by SASSY includes reconciling activity schemas such as the ones in Figure 6. Reconciliation may include interaction with users for confirming their willingness to have their activity schemas adjusted for compatibility. For example, Fred would be asked to consent to have his leaving (indirectly) known to the thermostat, while Bob would be asked to have the default message redirected to a different message targeted at the thermostat (a setT issued by the resolver). To avoid the repetition of such interactions at each space, users may annotate their schemas for consenting to any adjustments during reconciliation of the interactions with thermostats.

Part (d) of Figure 6 shows the general form of a value resolver. The resolver is annotated with a policy for accessing the service downstream, labeled by a lock, and with a resolution mechanism, labeled R. The access policies currently supported are P, for pseudonym, and A, for authorized access only. In the later case, user identities need to be registered beforehand with an access control mechanism to which the resolver initialization refers.

The two input messages ask (for a value V) and rem include an identification of the issuing user, U, to help maintain the correspondence between each request and withdrawal, to enable access control, and to enable some of the resolution mechanisms.

The value resolution mechanisms currently supported are least misery (LM), priority (Pr), majority (Maj), frequent users (Freq), and infrequent users (Infreq). Priority requires the resolver to refer to a classification of user priorities, i.e. to prior knowledge of user identities similarly to an access control mechanism. Majority takes the mode of the requests, adopting a tie-break policy initialized in the resolver, such as highest value, average, etc. Both Freq and Infreq keep statistical tallies of user requests, Freq giving precedence to the requests of frequent users, similarly to a frequent customer program, and Infreq giving precedence to infrequent users, similarly to the notion of fairness adopted in round-robin mechanisms. The stakeholders of the space may



(a) Fragment of SSS (b) Resulting design for Bob (c) Resulting design for Fred (d) Legend

Figure 7. Example of the effects of QoS preferences by different users in self-architecting and the resulting system design. In this example, Bob prefers low overhead at the expense of accuracy, resulting in a single service call; while Fred prefers accurate recognition, resulting in a system that combines the results of several recognition algorithms run in parallel.

choose the value resolution they deem appropriate and fair.

This example demonstrates how activity schemas can capture (a) user preferences with respect to setting values in situated services, and (b) the design of mechanisms to resolve differences in such preferences. It also illustrates how small variations in the interactions with such services can be reconciled dynamically.

5.3 Resolving Architectural Patterns

In addition to the functional aspects described so far in activity schemas, quality of service (QoS) plays a key role in distributed systems with constrained resources, as it is often the case in situated systems. For example, a service that would be a perfect fit from a functional standpoint may be deemed unacceptable by users due to low computing power, small screen, low battery, poor connectivity, etc.

SASSY optimizes QoS by automatically generating a number of candidate patterns for replacing what functionally constitutes each single service, and by choosing the pattern that best serves user defined QoS goals [4,25]. For example, for promoting availability, a service may be replaced by a redundancy pattern. Such goals are defined at the design level by highlighting paths of interest, so called Service Sequence Scenarios (SSSs), in an activity schema and annotating those with utility functions for the QoS aspects considered relevant by users.

What is new in multi-design is that each end user is enabled to associate *personalized* QoS goals for the *same* situated system, which is then reconfigured dynamically depending on which users are present. This is in contrast with current practices of software design, where it is assumed that stakeholders come to agree on the QoS goals for a system before deployment.

As a simple example, suppose that two users Bob and Fred work in shifts at an outdoors facility, such as a harbor or power plant. Their job includes carrying a wearable computer with a speech frontend to help them diagnose situations and create work logs. The schema for their activity includes several distributed services, such as a domain-specific database running on the company server, and a speech recognition service running on the wearable.

Aware that speech recognition is computationally intense, Bob is willing to articulate more clearly, disambiguating among candidate transcriptions if necessary, for the tradeoff of longer battery life and more computing cycles available for other tasks. For that he defined an SSS that includes the recognition service, Figure 7(a), and using interfaces such as in [26] he specified utility functions reflecting his preference. Fred also defined an SSS for the activity schema he shares with his coworker Bob, but instead he prefers to have high recognition accuracy despite the draw on computing resources.

The self-architecting algorithms in SASSY produce and deploy the design in Figure 7(b) whenever Bob dons the wearable, and the design in Figure 7(c) whenever it is Fred donning it.

This example illustrates how personalized QoS annotations in SSSs support system redesign at run time to optimally match the expectations of the user who is present at each given time. This is in contrast with current practice. Today, users need to agree on a common ground for QoS constraints, which then is used to guide self-architecting and deployment of the system.

6. DISCUSSION

A question not addressed in this paper is how to reconcile QoS goals when multiple users are present simultaneously. Referring to the last example in the previous section, what should happen if Bob and Fred share the equipment while working as a team?

Ongoing work investigates design notations for specifying the resolution of QoS goals following similar principles to the notation for resolving values, discussed in Section 5.2.

A different and harder problem concerns general mechanisms to reconcile desired features. The example in Section 5.1 illustrates features being added and shed from a situated system depending on which user is present. But what should happen when two users who disagree are present? Furthermore, how to recognize a disagreement in the first place? For example, if Fred's schema sends an output of service A to an input in B, and Bob's schema sends that same output to service C, do they disagree? Or should the output be sent to both service B and C?

Current software design notations include only positive statements, making it very hard to detect conflicts of desired features. Future work includes investigating notations for expressing undesirable features, which would make it possible to detect, as a pre-requisite to resolve, conflicts at the feature level. An important challenge for such extensions is making them both effective and usable by stakeholders and domain experts.

How to guarantee the scalability, security and privacy of service discovery is not specific to the proposed extensions, nor to SASSY, but a general engineering challenge for service discovery at run time. A promising approach is to combine results in the trust management body of work [27] with mechanisms similar to internet's Domain Naming System (DNS) for facilitating the interoperation of discovery mechanisms in different spaces.

7. CONCLUSION

The emergence of contemporary computing environments, such as smart spaces and pervasive systems, calls for rethinking the way software systems are currently designed, composed, and deployed. Many traditional principles and practices employed by the software engineering community are not applicable in this setting.

For example, location transparency, a common technique used to address the complexity of engineering distributed systems, is not a viable option in such setting. Moreover, establishing user requirements prior to system deployment is often not feasible, given that the users of such systems are often not known at design time, and if they are, their requirements may change at run time.

In this paper, we presented an approach that targets the aforementioned difficulties, and forms the centerpiece of SASSY. SASSY automates the composition and adaptation of service-oriented software systems. Through the use of an activity modeling language, the users specify functional, QoS, and spatial requirements of a situated software system. These requirements enable situated discovery of services, in which services are aware of, and announce their location, and service requests are scoped to a geographic area. The spatial requirements and situated discovery are in turn used by SASSY to automatically resolve conflicting preferences among the users of the system.

The traditional methods of engineering multi-user pervasive systems often force the users to make undesirable compromises, and arrive at a common, and often restrictive, set of requirements.

The work in this paper brings a paradigm shift. Not only it allows users to express and change their requirements at run-time, but it also exploits run-time properties of the system (e.g., location of users and services) to automatically resolve conflicting requirements. As a result, SASSY resolves conflicts in a more refined manner. Instead of enforcing an overly constrained set of requirements on the system (i.e., common denominator of all the users' preferences), it resolves them on a case-by-case basis and through the most effective means. We believe our approach also provides a more natural separation of humans' responsibilities: engineers develop services, ontologies, and infrastructure capabilities, while stakeholders specify system requirements in a more flexible and fluid fashion.

Avenues of future work include complementing our approach by incorporating temporal properties. This would allow us to resolve service providers not only based on the location of a user, but also based on the time of day, or potentially both. Our ongoing work also involves the development of the required monitoring facilities for enabling situated discovery in smart spaces, as well as a detailed empirical evaluation of the overhead and scalability of the approach in real-world situated systems.

8. ACKNOWLEDGEMENT

The work herein was funded in part by the National Science Foundation (NSF) under grant CCF-0820060. Any opinions, findings and conclusions expressed in this material are those of the author and do not necessarily reflect the views of the NSF.

9. REFERENCES

[1] A.G. Ganek and T.A. Corbi, "The dawning of the autonomic computing era," *IBM Systems Journal*, v. 42, 2003, pp. 5-18.

[2] F. Zhu, M. Mutka, and L. Ni, "Service Discovery in Pervasive Computing Environments," *IEEE Pervasive Computing*, vol. 4, Oct. 2005, pp. 81-90.

[3] V. Cardellini, et al., "Qos-driven runtime adaptation of service oriented architectures," *joint ESEC/FSE*, ACM Sigsoft, 2009, pp. 131-140.

[4] S. Malek, et al., "Self-Architecting Software SYstems (SASSY) from QoS-Annotated Activity Models," *Intl Workshop on Principles of Engineering Service Oriented Systems*, Vancouver, Canada: IEEE CS, 2009, pp. 62-69.

[5] Object Management Group, "BPMN Information Home," *http://www.bpmn.org/*.

[6] N. Esfahani, et al., "A Modeling Language for Activity-Oriented Composition of Service-Oriented Software Systems," *12th Intl Conf on Model Driven Engineering Languages and Systems*, Denver, CO: Springer LNCS, 2009.

[7] P. Ludford, et al., "Because I Carry My Cell Phone Anyway: Functional Location-Based Reminder Applications," *SIGCHI Conference on Human Factors in Computing Systems*, Montréal, Canada: ACM, 2006, pp. 889-898.

[8] A. Gershman, et al., "Situated Computing: Bridging the Gap between Intention and Action," *3rd Intl Symp on Wearable Computing*, San Francisco, CA: IEEE CS, 1999.

[9] J.E. Bardram, "Applications of context-aware computing in hospital work: examples and design principles," ACM New York, NY, USA, 2004, pp. 1574-1579.

[10] T. May, "El Camino wants paperless hospital," *San Jose Business Journal*, Apr. 2002.

[11] L. Barkhuus and A. Dey, "Is context-aware computing taking control away from the user? Three levels of interactivity examined," *5th Intl Conf Ubiquitous Computing*, Seattle, WA: Springer LNCS, 2003, pp. 159-166.

[12] J. Touzi, F. Bénaben, and H. Pingaud, "Prototype to Support Morphism between BPMN Collaborative Process Model and SOA Architecture Model," *Enterprise Interoperability III*, Springer, 2008, pp. 145-157.

[13] J.P. Sousa, et al., "Activity-oriented Computing," *Advances in Ubiquitous Computing: Future Paradigms and Directions*, IGI Publishing, 2008, pp. 280-315.

[14] J. Bardram, "From Desktop Task Management to Ubiquitous Activity-Based Computing," *Integrated Digital Work Environments: Beyond the Desktop Metaphor*, MIT Press, 2007, pp. 49-78.

[15] W3C, "Web Services Description Language (WSDL 2.0)," *http://www.w3.org/TR/wsdl20-primer/*.

[16] OASIS Consortium, "Universal Description, Discovery and Integration (UDDI)," *http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm*.

[17] H. Cai, Z. Shao, and A. Vaynberg, "Certified self-modifying code," *Conf on Programming Language Design and Implementation*, ACM, 2007, pp. 66-77.

[18] Paul Milgrom, "Auctions and Bidding: a primer," *Journal of Economic Perspectives*, vol. 3, 1989, pp. 3-22.

[19] S. Fatima, M. Wooldridge, and N.R. Jennings, "An agenda-based framework for multi-issue negotiation," *Artificial Intelligence*, vol. 152, Jan. 2004, pp. 1 - 45.

[20] H.R. Varian, "Economic Mechanism Design for Computerized Agents," *USENIX Workshop Electronic Commerce*, USENIX Association, 1995, pp. 13-21.

[21] G.S. Thyagaraju, et al., "Conflict Resolution in Multiuser Context-Aware Environments," *Intl Conf on Computational Intelligence for Modelling Control & Automation*, Vienna, Austria: IEEE CS, 2008, pp. 332-338.

[22] R. Hull, P. Neaves, and J. Bedford-Roberts, "Towards Situated Computing," *1st Intl Symp on Wearable Computers*, 1997, pp. 146-153.

[23] J. Hightower, G. Borriello, "Location Systems for Ubiquitous Computing," *IEEE Computer*, vol. 34, 2001, pp. 57-66.

[24] Smart Meter News Network, "Smart Meters," *http://www.smartmeters.com/*.

[25] D. Menascé, et al., "A Framework for Utility-Based Service Oriented Design in SASSY," *Joint WOSP/SIPEW*, San Jose, California, 2010, pp. 27-36.

[26] J.P. Sousa, et al. "A Software Infrastructure for User-Guided Quality-of-Service Tradeoffs," *Software and Data Technologies*, Springer CCIS, 47, 2009, pp. 48-61.

[27] A. Jøsang, R. Ismail, and C. Boyd, "A survey of trust and reputation systems for online service provision," *Decision Support Systems*, vol. 43, 2007, pp. 618-644.