Palpable Assemblies: Dynamic Service Composition for Ubiquitous Computing

Mads Ingstrup and Klaus Marius Hansen
Department of Computer Science, University of Aarhus
ingstrup@daimi.au.dk, klaus.m.hansen@daimi.au.dk

Abstract

An important characteristic of ubiquitous computing is that the computational services in our environment are envisioned to be far more interconnectable than today. This means it should be possible to combine them to suit the purpose at hand at any given time. However, given a particular combination of services, there are numerable combined behaviours that could be meaningful. Furthermore, ubiquitous computing is often characterized by dynamically changing, heterogeneous combinations of services. It is therefore necessary to be able to specify the desired behavior and to be able to switch dynamically and easily between several such behaviours. Assemblies, as a concept in ubiquitous computing, has been suggested as a mechanism to represent a set of services and their behavior.

After summarising the use-inspired notion of an assembly, this paper (1) clarifies the concept from a software engineering perspective, arguing that an assembly is both an architectural connector and a component (2) that an assembly should have a programmatic representation (3) provides a discussion of what it should comprise, based on realistic examples grounded in participatory design (4) the challenges facing the adoption of the concept, and finally (5) shows that a particular suggestion for its implementation based on publish/subscribe is architecturally realizable.

1. Introduction

When the way we use computers now seems well on its way towards the vision of ubiquitous computing [21], it can partly be ascribed to the hardware that makes it possible in the first place. Stationary computers account for only a fraction of the microprocessors in use [22]—the by far largest share are embedded processors in cars, dishwashers, TVs, cellphones, DVD players and other appliances. Why, then, does it not feel quite like we live in the world Mark Weiser described back in '91 ?

An important reason is arguably that the true potential of all these microprocessors is only realized once the software running on them becomes interconnected. Instead of a device being just a cellphone, it should be possible to use it as a generic means of connectivity, being combinable with a camera so that pictures are automatically uploaded to a server at the home office. The camera should be combinable with the TV, and the TV, in turn, should be applicable as a general purpose display.

The notion of an assembly has been proposed to describe such dynamic combinations of devices and services in the PalCom project[1]; as such it embodies both software and physical concepts of composition and de-composition [15]. In particular, it supports composition of heterogeneous services and highly dynamic changes in which services are available; properties which are characteristic of ubiquitous computing.

The purpose of assemblies has primarily been that of *describing* the use of technologies. Therefore an assembly is a use-oriented concept: a combination of computational devices or services that is considered a conceptual whole by its user. The subject of this paper is how *assemblies* should be realised programatically. Specifically, the contribution of this paper is twofold:

First, to discuss the abstractions used to comprehend such combinations of devices, assemblies, as are inherent to pervasive and ubiquitous computing. People, whether users or programmers, conceive of and understand the world using concepts [8, pp. 279] and the aptness of the concepts determine how easily something is made sense of. Focusing on the realisation of assemblies in software, the main perspective here is how programmers should understand assemblies. To that end the concept is clarified by relating it to the architectural abstractions of components and connectors. The proposition is that assemblies, conceptually, are both.

Second, it is argued that the concept of an assembly should be realised through an explicit programmatic construction that represents it at program-time and at runtime. The success of object orientation has already established the advantage of using the same concepts to express programs as are used to think about their problem and application domains. The intuition behind a programmatic representation for assemblies is to use similar constructs to express assemblies as are used to think about them.

To demonstrate both this point and the feasibility of

[1] http://www.ist-palcom.org

mixing components and connectors, an architectural prototype [4] has been implemented. It exemplifies assemblies and shows that the idea of a combined component and connector is architecturally realizable, at least when using topic-based publish/subscribe for intercomponent communication. The publish/subscribe communication paradigm is a special case of event-based communication in which events are routed from publishers to subscribers based on specific forms of subscriptions [7]. In topic-based publish/subscribe, events are published to particular named topics and routed to subscribers which have subscriptions matching the topic. In general, using publish/subscribe in ubiquitous computing is interesting in that it provides a flexible and adaptive communication mechanism among others through support for time, space, and flow decoupling.

This use of publish/subscribe introduces some issues related to scoping which can, however, be addressed in a way that is consistent with the proposed form of the assembly concept.

The next section introduces the concept of an assembly. Section 3 explains why a programmatic construct corresponding to the concept is needed. Section 4 discusses how to implement something that is both a component and a connector and describe the architectural prototype of the chosen approach. Section 5 reviews related work, section 6 addresses the challenges which set the present state of the art apart from realising the envisioned scenario described next. Section 7 concludes the paper.

2. The Assembly Concept

This section presents the concept of an assembly through a prototypical example followed by an elaboration of the concept. The scenario is used as a running example in the remainder of this writing. One note on terminology: "component" is used in its normal sense, and by "service" is meant *runtime component*.

2.1. Example: The SitePack Scenario

As part of the PalCom project [15], workshops have been conducted workshops with landscape architects, sociologists and computer scientists to discern areas that lend themselves well to support by pervasive computing [5].

The workshops were conducted with participatory design [6] techniques such as mock-ups sessions and roleplay. This speaks for the scenario as a potentially realistic one.

Scenario. *Whenever deciding where in the landscape a wind farm should be placed, landscape architects are hired to determine its visual impact in the area surrounding a potential location. They need to map what percentage of the wind farm is visible from which areas.*

The first step in doing so is to compute a ZVI (Zone of Visual Influence) map, where the red zones *are the areas where the most of the wind farm is visible. Trees and other obstacles are not taken into account when calculating the ZVI maps from height maps of the landscape. Therefore landscape architects have to go to the red zones and take pictures of the environment to get a feel for the actual visual impact.*

There are often roads running through the red zones, so the landscape architect drives along these and has to simultaneously keep track of when the car enters a red zone, of driving the car and of keeping an eye on the landscape. Therefore it would be useful to automatically receive notifications on the location relative to the boundary of the red zone. That can be done by combining three devices: GPS, display, and some node holding a ZVI service.

A second task arises, when the red zone is entered the landscape architect gets out of the car, walks around and takes pictures of the potential wind farm location. The location and perhaps orientation of the camera when taking these pictures should be recorded along with the pictures. For that task a purposeful combination of devices would include a GPS receiver, a Camera, and a cell phone. The cell phone provides connectivity to a server at the office, to where the pictures are uploaded after having been indexed with position information.

We have found many tasks like these two that can be supported by combining a set of "core" devices.

2.2. An assembly is both component and connector

The notion of assemblies described so far is: some combination of devices or services that is meaningfully considered a cohesive whole by the user. This does not address how assemblies are realised in software. A refinement of the abstraction towards that purpose will be given in this section.

First consider the running example: That an assembly must be a *dynamic* construct is evident from the scenario because the constituent devices should not be permanently bound in the assembly—it should be possible to detach the cellphone and use it as a regular phone when not taking pictures. Similarly, while driving to and from the photographed site, the GPS receiver should form part of the navigation system in the car—another assembly.

The assembly for auto-uploading pictures to a server (the second combination in the scenario) let us call it PhotoAssembly, has the *roles* Camera, Position and Connectivity. These roles define what types of services are required to use the assembly. This is shown in figure 1 with a notation inspired by Kristensen [12]. The level of generality implied by the role-names is intended, since any compatible services capable of providing position informa-

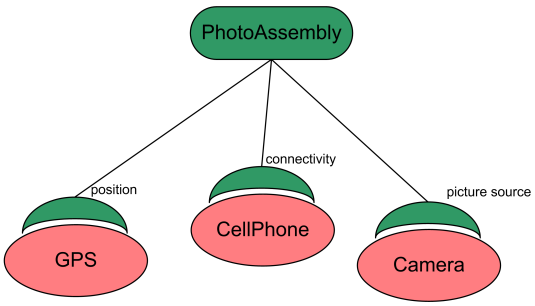

Figure 1. The PhotoAssembly bound to three (thus constituent) services—the oval shapes. The notation is inspired by Kristensen [12].

tion could in principle be used in the assembly, and the same goes for the connectivity and the camera roles. In the concrete situation of the scenario, the roles are played by the services residing on the Camera, GPS-receiver and Cellphone respectively. In this description the assembly appears an architectural connector, since it binds several components together.

The assembly mentioned first in the scenario—the one that notifies the driving landscape architect of the car's location relative to the red-zones—could in fact be modelled best as two assemblies: One containing a GPS service on a node and a ZVI service on a node that has a display and that can be queried for the distance to the nearest red-zone and show the current position with the ZVI service. And another assembly, containing the first as well as a service that can transform textual notifications into speech so the driver can hear them. This model is shown in figure 2. However this scheme requires the assembly to be something that can play a role in a connector: a component.

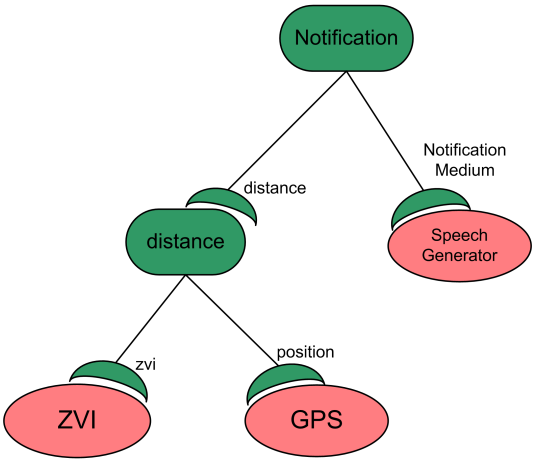

Figure 2. Using two assemblies to model the setup of the first scenario.

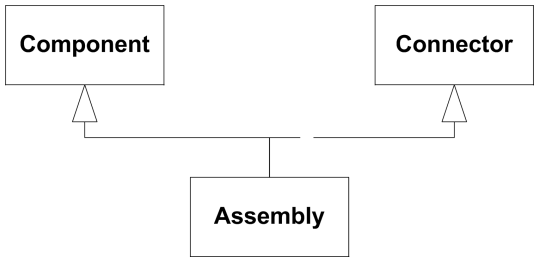

Figure 3. The relationship between assemblies, components and connectors.

This illustrates a key point of this paper: that an assembly is, conceptually, both a component and a connector, see figure 3. That conjecture requires more rigorous argumentation than the illustrative example above. To that end several non-obvious points must be established: (1) That it makes sense to distinguish the interaction/functionality aspects in assemblies and use the abstractions component and connector to adequately capture them (2) that an assembly must necessarily be both (3) that an assembly by being both does not, conceptually or by implementation, ignore the point made in (1). The first two points are addressed below. The third point is deferred to section 4 which presents how the construct have been implemented.

Interaction and functionality are distinct. This is a point that historically has been recurring in many forms. Components have often been the only implementation construct—hence we speak of component-based software rather than component- and connector-based software. However connectors, which mediate interactions and engender a protocol, deserve their own abstraction [17] and what exactly they represent has been formalised [2], and implemented as a first class construct in e.g. ArchJava [1].

Other forms in which the distinction made with components and connectors have been argued include computation/coordination [10], algorithm/interaction [17] and objects/associations [12]. Further, the formalisms appropriate to each are different: Turing machines [13] underlie our notion of algorithms, but are incomplete for specifying interactions [20], for which e.g. the π-calculus was designed [14].

Assemblies abstract both interactions and functionality An assembly should have both an algorithm and a protocol. An algorithm is a recipe for realizing some functionality. A protocol, however, is a set of rules specifying how interactions take place.

Because the assembly must realize a particular behaviour of a given combination of services, it often needs an algorithm. First, not all desired behaviours can be realized

by simply making the constituent services interact—it is in general not possible to anticipate all future uses of an information appliance [18]. Second, even in the cases where the participation of a service in an assembly can be predetermined, some parts of the realisation of this behaviour may not properly belong to any of the constituent services. This latter point is addressed again in the next section.

An assembly must engender a protocol because it binds together several services so that they interact. The protocol serve two purposes. It (1) handles the coordination between the services, and (2) specifies routing of data between them.

This concludes the argument at the conceptual level that an assembly is both component and connector. We now turn to a discussion of the consequences this have for their realisation in software.

3. Programmatic Representation for Assemblies

An assembly should have an explicit programmatic representation at runtime and program-time because:

- *The definition of an assembly should be localised* Keeping related things together is good software engineering: individual parts of a system should be internally coherent. The behaviour of an assembly and the interactions supporting it form a conceptual whole.

- *Some information about the assembly does not have a proper home in any of its constituent services* This is often the case with exception handling. Exceptions should be handled in the context where they make the most sense. For instance, if the signal of the GPS receiver is blocked, this could give rise to an exception. However, in the context of the photo assembly this could meaningfully be handled by informing the user. In another context, such as if the GPS receiver were on board an autonomous vehicle, the same exception should be handled in a quite different way, maybe by halting the vehicle and switching to an auxiliary navigation system.

- *Assemblies should be independent of other components and connectors at runtime* The set of services that play a role in an assembly is likely dynamic. The responsibility of managing a flow of alternating services does not belong with any of the delegate services as this is a changing set. Instead it belongs to the assembly. Therefore the assembly should have an existence of its own at runtime, separate from the delegate services.

- *Assemblies can be stored away, and later reactivated* The scenario illustrate that the combination of devices and their aggregate behaviour should be reusable from situation to situation. Relative to the scenario, the PhotoAssembly would be stored on e.g. the camera, remaining inactive while the landscape architect is underway in the car, and then activated upon arrival to the site of the planned wind farm.

Additionally, an explicit representation of assemblies enable the use of OO relationships to model and construct them:

- *Assemblies should be specializable* One assembly can specialise another. For instance, the photo assembly could be formed using a more general assembly, PictureIndexer, that index pictures with a string of text. This means that assemblies can be abstract as well, where abstraction is meant in the traditional sense of not being instantiable.

- *Assemblies should be composable* Since an assembly is a service itself, it can play a role in another assembly. This is really just standard composition.

4. Prototyping the Assembly Construct

This section describes how the proposed assembly construct have been evaluated. Since it is a programmatic construct, it is evaluated by implementing it and applying it to the example with the PhotoAssembly. The implementation is in the form of an architectural prototype [4]. The prototype has been implemented in the Ruby programming language [11]. ArchJava [1] was also considered because it provides language support for connectors. However Ruby was favoured because it is a dynamically typed language which is more in correspondence with the dynamic nature of assemblies, and allows easy simulation of and modification to the behaviour of various language mechanisms.

Communication between services is done with publish/subscribe. Using the publish/subscribe paradigm means that the devices themselves need not be simulated, since the boundaries of the services are where distribution is visible in the programming model: Inside components, normal procedure calls can be used, but only publish/subscribe can be used between components. Distribution of components is thus transparent, but clearly in a sense different from an RPC communication model which attempt to hide distribution altogether. For that reason devices were not simulated explicitly, as it would have altered nothing in the design or implementation of the services in the example. The point argued is not that publish/subscribe is appropriately the only means of intercomponent communication. Only that it is appropriate to illustrate what assemblies are because it is a very basic form of interaction.

An assembly is implemented as a connector, and a component. The component is bound permanently to a special *behaviour* role of the connector. This role is special since (1) its port is responsible for handling contingencies related to the function of the assembly, (2) it determines where the assembly is deployed and thus gives form to it and (3) it is always bound to the assembly. The assembly provides both ports and roles, and is logically equivalent to what is illustrated in figure 4. Note that the figure does *not* illustrate deployment view, only the logical view. Deployment is intuitively more consistent with the notation in figures 1 and 2, since the behaviour of the assembly is deployed on some node, and the roles are co-located with the services to which they are bound. This is shown in figure 5. The type of network connectivity underlying the publish/subscribe primitives used is not shown. We assume the services to be running in a Palpable Runtime Environment which is currently being development in the PalCom project; it includes a virtual machine with primitives supporting publish/subscribe.

As a connector the assembly has a set of roles. Each role acts as a message filter, specifying which publish /subscribe topics allow a message to pass through. Two patterns of filtering were found useful: a message is either not mentioned in the role, or it is in the IN-set or the OUT-set of that role. When a topic is in he IN-set of a role, the service which is bound to that role will only be allowed to receive messages on that topic when they are published by a service that is a member of the same assembly. In the example of the PhotoAssembly shown in figure 1 the behaviour role would have the topic *position* in its IN set, because those coordinates that it obtains from the GPS that is part of the assembly are the only ones that should end up in the pictures.

Conversely, when a topic is in the OUT-set of a role, any messages published on that topic by the component bound to the role will only reach subscribers that are in the same assembly. In the Notification of figure 2, the behaviour role would have the notification messages in the OUT-set, because only the chosen notification medium, the SpeechGenerator should be used, while still allowing it to be used by other assemblies.These two filtering-patterns were sufficient for the PhotoAssembly, and though perhaps not fine-grained enough in other cases, it illustrates what a role can be and that is may have some programmatic substance.

To activate an assembly, services must be bound to its roles; Section 5 outlines approaches for matching services to assemblies.

5. Related work

Fujii & Suda [9] present an approach to dynamic service composition that relies on automatic composition of services based on a high-level expression of a goal such as “print out direction from home to restaurant”. This approach requires an infrastructure to deliver the deducted service configuration. The assembly approach, conversely, embrace the highly distributed and ad-hoc nature of pervasive and ubiquitous computing, and does not depend on computationally intensive deductions. An assembly can be carried on a device, and become active across devices when its required services are within communication range.

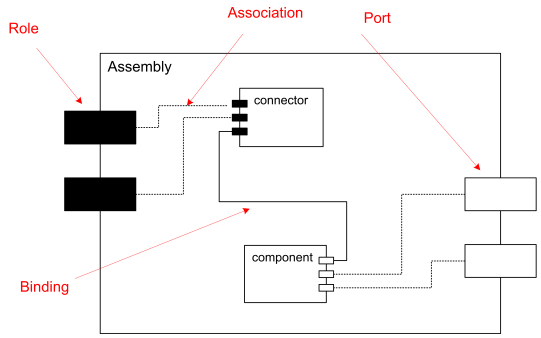

Figure 4. The logical implementation-relationship between assemblies, components and connectors. The assembly encapsulates its parts, exposing only their “vacant” roles and ports. The black rectangles represent roles of a connector, the white are ports of a component.

Ponnekanti et al. [16] also recognize the need for supporting dynamic combination of services to e.g. combine a camera with a printer. They describe ICrafter, a framework providing infrastructure support for a class of ubiquitous applications. It supports aggregation through automatic generation of user interfaces for combinations of services. The *generators* which produce the interfaces are generic software entities relying on general programmatic interfaces that the constituent services implement. The assembly approach is similar because it address the same general problem of combining services, but it applies to a broader class of ubiquitous computing systems because assemblies do not assume a supporting infrastructure outside of the constituent devices.

Emerging web service orchestration approaches such as Business Process Execution Language for Web Services (BPEL4WS) [3] provides abstractions for defining service interaction and composition in Service-Oriented Architectures (SOAs). Being a SOA concept web service orchestration typically works on higher-level services than do assemblies. Typically, approaches such as BPEL4WS require a central process management component, but conceptually and with respect to implementation the concept of explicit and distributed assemblies could also be relevant in this context.

The central purpose of the Java-based Jini framework [19] is to create “federations” of services that may

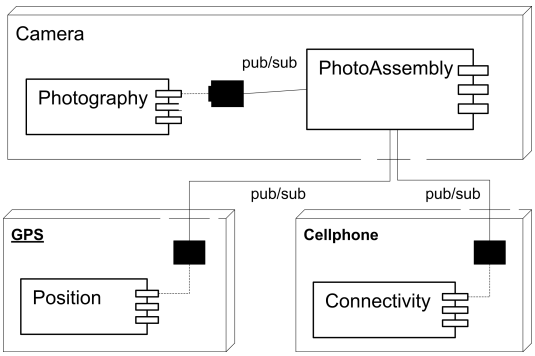

Figure 5. A deployment diagram corresponding to figure 1. The associations between the assembly and its roles are of the same type as those shown in figure 4 and thus encapsulated in the connector part of the assembly.

jointly implement functionality. As such Jini is suitable for many pervasive and ubiquitous computing applications. However, Jini does not have an explicit and dynamic representation of federations at runtime. Incorporating the idea of assemblies into Jini might be a possibility.

6. Challenges and Future Work

Up until now our line of argumentat has been presented in a perhaps optimistic spirit. However, the following challenges still need to be adressed:

Handling Heterogeneity The scenario described initially cannot be realised with ease using todays devices, the hard- and software of which would have to be prepared for interaction with other devices. The current trend goes toward that with e.g. the increasing support for Bluetooth in various devices. Regarding the heterogeneity of software, it is clear that components on these devices are likely to be written in different languages, currently perhaps .NET or Java.

Gelernter & Carriero [10] point out that the separation of interaction (coordination) from computation is favourable from this point of view, since a standardisation can thus be limited to a coordination language which is largely orthogonal to a computation language [10]. Their scheme is that any intercomponent-communication could be achieved using a coordination language (e.g. Linda) plugged into the language used to program algorithms. That idea matches assemblies well, and in the architectural prototype outlined earlier, publish/subscribe is used instead of Linda, which is achieved by adding (implementing) the simple primitives *publish(msg)* and *subscribe(msg)* to Ruby.

Dependence on service discovery One of the things not adressed in the prototype implementation is how the services are found and subsequently bound to an assembly. This clearly relies on service discovery (SD). The assumptions currently made about SD is that there exists some way of specifying a service such that certain assumptions hold about how it can be used. Therefore discussion of *typing* in relation to assemblies has been adressed only implicitly.

Further empirical exploration of the assembly concept A final concern to be adressed in future work is evaluating and exploring the assembly construct further. First, by applying it to more varied use-situations to establish its general applicability/limitations. Second, using it for prototypes that are tested with real users, rather than the architectural prototype taken as a first step. Issues at the use-level include: how may services be shared appropriately and dynamically seen from a social and interaction perspective? How autonomous are assemblies to be in use? What is the relationship between physical assemblies and the software constructs supporting them?

7. Conclusion

The notion of an assembly from [15] was summarised. It was refined toward a software engineering understanding of its meaning: that it should be both a component and a connector, and have an explicit representation at runtime and at compile time. A scheme for its implementation was presented, and a brief description of an architectural prototype of the concept was given. Finally, the assembly approach was related to existing approaches documented in litterature, and the challenges and future work of the construct was presented.

8. Acknowledgements

The research reported in this paper was partially funded by ISIS Katrinebjerg and by the PalCom IST project. We thank Monika Büscher for organizing the workshop on assemblies and for commenting on this paper. Henrik Bærbak Christensen for discussions and comments on earlier drafts of this paper. Jonas Lövgren, also for commenting on the paper, and Erik Ernst, Jakob Bardram, Peter Ørbæk, Michael Christensen and the rest of the participants in the PalCom project for discussions of the assembly concept.

References

1. J. Aldrich, V. Sazawal, C. Chambers, and D. Notkin. Language support for connector abstractions. In L. Cardelli, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'03)*, volume 2743 of *Lecture Notes in*

Computer Science, pages 74–102. Springer Verlag, July 2003.

2. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

3. T. Andres, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business process execution language for web services. version 1.1. Technical report, BPEL4WS Partners, May 2003. Available from ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf.

4. J. Bardram, H. B. Christensen, and K. M. Hansen. Architectural prototyping: An approach for grounding architectural design and learning. In J. Magee, C. Szyperski, and J. Bosch, editors, *Proceedings of the Fourth working IEEE/IFIP Conference on Software Architecture (WICSA'04)*, pages 15–25. IEEE Computer Society, 2004.

5. M. Büscher. Vision in motion. *Environment and Planning A*, 2005? (forthcomming).

6. P. Ehn and M. Kyng. Cardboard computers: Mockingit-up or hands-on the future. In J. Greenbaum and M. Kyng, editors, *Design at Work: Cooperative Design of Computer Systems*, pages 169–195. Lawrence Erlbaum Associates, 1991.

7. P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

8. M. W. Eysenck and M. T. Keane. *Cognitive Psychology*. Psychology Press, Tayler & Francis Group, 27 Church Road, Hove, East Sussex BN3 2FA, 4^{th} edition, 2000.

9. K. Fujii and T. Suda. Dynamic service composition using semantic information. In *Proceedings of the 2nd International Conference on Service Oriented Computing*, New York, New York, U.S.A, Nov. 2004. ACM.

10. D. Gelernter and N. Carriero. Coordination langages and their significance. *Communications of the ACM*, 35(2), 1992.

11. A. Hunt and D. Thomas. *Programming Ruby: The Pragmatic Programmers Guide.* Addison-Wesley Longman, 1 edition, 2001. Available from http://www.ruby-lang.org.

12. B. B. Kristensen. Associative modeling and programming. In *Proceedings of the 8th International Conference on Object-Oriented Information Systems*, Montpellier, France, 2002. Springer-Verlag.

13. H. S. Lewis and C. H. Papadimitriou. *Elements of the theory of computation*. Prentice-Hall, 2 edition, 1998.

14. R. Milner. Elements of interaction: Turing award lecture. *Communications of the ACM*, 36(1):78–89, Jan. 1993.

15. Palpable computing: A new perspective on ambient computing. annex i: Description of work.

16. S. R. Ponnekanti, B. Lee, A. Fox, P. Hanrahan, and T. Winograd. ICrafter: a service framework for ubiquitous computing environments. In *Proceedings of UbiComp 2001*, volume 2201 of *LNCS*, pages 56–75. Springer-Verlag, 2001.

17. M. Shaw. Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. In *Proceedings of Workshop on Studies of Software Design*, Lecture Notes in Computer Science. Springer-Verlag, 1993.

18. L. A. Suchman. Practice-based design of information systems: Notes from the hyperdeveloped world. *The Information Society*, 18(2):139–144, 2002.

19. SUN. Jini technology core platform specification. version 2.0. Technical report, SUN Microsystems, 2003. Available from http://www.sun.com/software/jini/specs/core2_0.pdf.

20. P. Wegner and E. Eberback. New models of computation. *The Computer Journal*, 47(3):4–9, 2004.

21. M. Weiser. The computer for the 21st century. *Scientific American*, 13(2):94–10, Sept. 1991.

22. S. Wong, S. Vassiliadis, and S. Cotofana. Embedded processors: Characteristics and trends. Technical report, Computer Engineering Laboratory, Delft University of Technology, 2004.