Applying Direct Combination to afford spontaneity in pervasive computing

Simon Holland
Department of Computer Science, The Open University, Milton Keynes, UK.
s.holland@open.ac.uk
Technical Report TR2002/11

A version of this paper was accepted as: Holland,S., Gedenryd,H., and Morse,D. (2002). Applying Direct Combination to afford spontaneity in Pervasive Computing. UBICOMP 2002 Workshop on Supporting Spontaneous Interaction in Ubiquitous Computing Settings, Gothenburg, Sweden.

Abstract
In rich pervasive environments, there will be numerous opportunities for end users to dynamically create services of interest by causing *two or more devices or resources to interoperate together,* often under changing circumstances. In general, users find this kind of process hard to manage. Existing programming architectures make the situation difficult to address in a principled, scaleable way. Users find it hard to tackle such problems via devices with small, resource-poor user interfaces. It is proposed that a good theoretical basis for addressing an essential aspect of all of these problems is the theory of *Direct Combination*. When the Direct Combination framework, based on the theory, is applied to spontaneous interactions, the user interface can be made relatively simple, and the amount of search required by the user to specify desired actions can be greatly reduced. We present Direct Combination (DC) and the new interaction techniques it gives rise to for pervasive environments. We consider two different support architectures. We argue that one of these, the role-based architecture, has particularly good properties for modelling rapidly changing pervasive environments, and for highly distributed implementations. We demonstrate how the concept of viewpoints can be used to focus, filter and afford operations, and how this can be well supported by the role-based architecture.

1 The problem

There is general agreement that mobile and ubiquitous networked devices, on the person and in the environment, will proliferate and diversify widely. This will make a very wide range of services and activities available to users. In sufficiently rich environments, there will be numerous opportunities for end users to access, or to dynamically create, services of interest by causing *two or more devices or resources to interoperate together,* often under temporary circumstances, and with unforeseen combinations of functionality emerging [1,2,3]. A simple, relatively non-problematic example is that is that a user with a PDA in an unfamiliar place might wish to show another user a document on a nearby screen. In general, users find impromptu interoperation of two or more resources in changeable circumstances hard to manage [3, 4]. Existing programming architectures tend make this kind of task inconvenient, since the necessary functionality is generally controlled via application programs, which cannot take into account all possibilities, and which depend on low level system resources such as drivers, that are generally difficult for end users to configure [2]. Whenever two or more distinct resources are involved, the problems multiply combinatorially. Consequently, improvising solutions to such tasks often involve the user in non-trivial searches of the user interface. The problems are particularly acute when the search is performed on the move via mobile devices with small, resource-poor user interfaces. Typically users are forced to spend time and attention distracted from their main task, searching for needed functionality in a sequence of small screens or menus. The problem of supporting interoperation in changing circumstances, especially in pervasive systems has been called the problem of *spontaneous interaction*. There are currently no well-developed accepted solutions to the problem of spontaneous interaction. Two key issues are the lack of interaction techniques well suited to the needs spontaneous interaction, and the apparent mismatch of the standard application program architecture with the needs of pervasive computing.
We believe that it is vital to develop approaches that allow the user to specify what they want as simply and directly as possible, while at the same time making it possible to take full advantage of whatever human and machine knowledge is available. Of course, other approaches are equally worthwhile. But from an HCI point of view, we identify two key problems: to find the most economical means for the user to specify what they require in a spontaneous interaction; and to find programming architectures capable of supporting the required simplicity.

2 Proposed solution

We propose that a good theoretical basis for addressing both of these problems is the theory of *Direct Combination* [5,6]. When the Direct Combination framework, based on the theory, is applied to spontaneous interactions, the user interface can be made highly economical, and the amount of search required by the user can be greatly reduced. The theory of Direct Combination, with its associated interaction techniques and architecture is perhaps best introduced by means of an example.

2.1 Scenario 1

Anne, on holiday in China, wanted to display a file from her PDA on the wall display at the meeting place in the shopping mall, for her companion Bill. She searched the control panels and menus of her PDA to no avail. There didn't seem to be anything relevant in any of the word processor menus, nor in the screen set up menus, nor in the communications or utility menus for communicating with any kind of external display, let alone this one. Anne managed to locate and download a universal remote control program [7] for the wall display. However, searching the various operations for the wall display remote control, nothing from the endless options seemed relevant to displaying a file from her PDA. Suddenly, Anne remembered that her new PDA was DC compliant. Anne selected the file on her PDA and then zapped the wall display screen (i.e. pointed the infrared id-tag reader on the PDA at the wall display screen). Now her screen offered her the single action applicable to this particular pair of resources*:*

Display the selected file on the screen

(In the general case, several options would be offered, but in this case, only one action happened to be applicable). Anne accepted the action and the wall display displayed the file. Anne then used the universal remote to adjust the display as she wanted it. When she had finished showing the document to Bill, Ann used the task monitor on her PDA to find the task and break the connection.[1]

A key principle of the Direct Combination framework as illustrated by this scenario is that if the user is allowed to indicate in advance *two or more interaction objects* involved in an intended action, then, given the appropriate architecture, the system can use this information to constrain significantly the search space of possible commands. This allows the system to present to the user a space of focused relevant options to choose from, instead of the unrestricted space of commands. This kind of interaction is known as *pairwise* interaction, which together with conventional *unary* interactions and *n-fold interaction* are all special cases of the general DC interaction pattern. The pattern with two interaction objects is particularly useful, but zero or more objects are the general case. For example, when Anne initially zaps the wall display by itself to find relevant actions, this is actually an example of a *unary* interaction (though Anne need not know this to make use of it effectively). User interfaces built using the Direct Combination framework give the user the freedom to specify the parts of commands in any order desired, for example *noun noun* (to be followed later by a verb), as well as the more conventional *noun verb*, or even the simple *verb,* any of which may be modified or followed by arguments. Crucially, at each stage, feedback is given on how choices made so far constrain further choices. One important aspect of these new interaction styles are that they are entirely compatible with pre-existing interaction styles (e.g. the predominant unary interaction *noun verb*, and the simple *verb*). All of the patterns are available for the user to use whenever he or she thinks fit.

The theory of Direct Combination (DC) analyses the affordances of incomplete command patterns from the perspective of *search*, as informally illustrated above. From the theory, it is relatively straightforward to derive:

- new *interactions styles* based on the range of *interaction patterns,*
- *principles* for applying those interaction styles,

[1] In order to keep the scenario simple, we have undersold the amount of help that Direct Combination could give Anne. For example, to find out what the wall display could do, Anne might have zapped the wall display by itself (a unary interaction), and been offered a variety of commands and queries suited to the display. If the unary protocol to the display was capably designed, then one of the first options might reasonably be to download a suitable universal remote. In any case, in order to get a more targeted range of options, Anne could have made a pairwise interaction between the *display* and the *PDA itself* by zapping the display and using the *'include self'* button to explicitly include the PDA in the interaction. Several similar interactions can be found in [6]. The choices would then be expected to include the following options:

- *mirror screen contents on display until further notice,*
- *show current contents of screen on display,*
- *download universal remote for display.*

• a new architecture to support the interactions,
• a new approach to *domain analysis* to populate the architecture.

The principles, interaction styles, architecture, and analysis techniques taken together are referred to as the DC *framework*. The framework has already been applied successfully to desktop computing [5] and in early prototype to pervasive computing [6]. The theory predicts that, for tasks in sufficiently rich environments (such as environments where spontaneous interactions are frequent) the DC framework will have the capacity to reduce the user's search space. Other things being equal, this reduction in search space would be expected to lead to a reduction in the users' time and attention expended on distracting search tasks. Consequently, we predict that, for environments where spontaneous interaction is commonplace, the application of DC techniques to ubiquitous environments has the capacity to speed up task performance and to reduce mental load.

3 Direct Combination theory

Having introduced DC, let us now consider the DC framework, including the underlying architecture, in a little more detail. The Direct Combination theoretic model analyses the space of user commands as constructed from elements such as *noun, verb* and *qualifier*. The theory identifies a strong constraint; imposed by most current user interfaces, that limits the order in which the elements may be assembled. Typically a user must specify a noun first, e.g. by selecting an object (or exceptionally a collection of nouns, all to be treated alike), followed by a verb, and finally a set of qualifiers. This restriction would be immaterial if it did not have profound implications on the *search strategies* afforded to the user. As a user incrementally specifies each element of a command pattern (e.g. by zapping an external entity, or selecting a virtual object or menu item), a good user interface should continually provide *feedback* on how the choices made so far afford or restrict further choices. However, where a selected entity affords numerous rich interactions (i.e. has many applicable operations), if the user has been forced to select a noun first and then inflexibly to follow it with a verb, the user may be left with a large space of possible operations (verbs) to search. This is precisely the problem faced by Anne in our scenario before the remembered that she could use Direct Combination. The same problem occurs in most current user-centred (as opposed to inference-centred) approaches to spontaneous interaction. Where each individual resource is complex or unfamiliar, or the names or classifications of tasks are unknown to the user, searches of the user interface tend to disrupt flow and to impose unwelcome diversions on the user.

3.1 Freedom of interaction afforded by the Direct Combination framework

The principles arising from DC theory propose that the restrictions on search outlined above should be dropped, and that users should be given the freedom to assemble commands patterns in any order they like. This freedom is of particular importance for spontaneous interaction, since by indicating the relevant objects of interest, the search space for relevant commands can be greatly constrained, as illustrated above. In sufficiently rich environments, application of the DC framework has great potential to reduce the user's search space. An additional benefit is that in environments rich in objects of interest, users often know, or can recognize, what objects they want to use (a screen, a wallet, a car, a room, a document, etc). However, operations, apart from commonly used operations, tend to be relatively more abstract, and harder for people to recall by the name used in a system. The freedom given by Direct Combination allows users to take advantage of *recognition* (easy) vs. *recall* (hard).

3.2 Is DC an essential component of any user-centred approach to spontaneous interaction?

More generally, the theoretical model of DC suggests that in sufficiently rich environments, any user-centred approach to spontaneous interaction which does *not* give user the freedom afforded by direct combination will tend to impose uneconomical search strategies on users. For this argument to have compelling force, it would be necessary to show:

• empirically that DC reduces search, reduces time spent, reduces mental load,
• that DC interaction styles can meet the general requirements for any interaction style for spontaneous interaction, and has reasonable generality and power,
• the feasibility of the architectural support required.

The empirical work we are carrying out on DC will be reported elsewhere. We will focus here solely on the latter two points, starting with the architectural support required by DC.

4 Architectural support for Direct Combination

Most systems as currently constructed have no way of computing on the fly what operations are relevant to a collection of two or more selected objects, even in an isolated desktop system, far less in a distributed ubiquitous

environment. This is a requirement for any architecture that supports DC. There is no doubt an indefinite variety of architectures that could support the DC framework in pervasive environments, but we will focus on the pros and cons of just two such architectures, inheritance-based vs. role-based, that we have implemented and applied to DC. Either architecture can in principle be applied in at least three different ways in a pervasive environment, with certain provisos, as follows.

a) **Thin client:** thin, small cheap combination clients supported by servers: this version would be well suited to infrastructure rich environments such as present day offices.

b) **Thick client** thicker combination clients that carry their own object models, databases and the functionality required to request, and in a few cases carry out, operations between other objects: this variant would be well suited to poorly resourced outdoor environments.

c) **Fully distributed:** more realistic pervasive environments of the future, where the representation of the environment is more distributed: some sources will hold information about available entities, and others serve as sources for deeper information about the capability of entities that may appear, for example in manufacturers databases.

4.1 An inheritance-based architecture for Direct Combination

Our first architecture (inheritance based) is relatively simple, and can generally be created by relatively simple modifications to existing object systems. This was the architecture used in earlier work, which applied DC to desktop computing [5]. Essentially the same architecture can also be applied to pervasive computing with some provisos [6]. This architecture exploits the structure inherent in polymorphic operations and object-oriented inheritance hierarchies, to make information about possible interactions between resources relatively easy to specify, propagate and compute. Code encapsulating such information is added at points as high as possible in existing abstraction hierarchies in order to specify what interactions are possible and how they are carried out. This information may be overridden in leaf classes where necessary. This architecture can be deployed relatively easily in either thick- or thin-client versions, but is less practical to apply to fully distributed pervasive environments. With any kind of inheritance-based hierarchy, it can be hard to keep the system well-factored when new kinds of resources are being continually introduced whose functionality crosscuts previous well defined class boundaries. This is a characteristic of pervasive environments. When using an inheritance-based solution in such situations, there tends to be a need for continual refactoring at best, or the build-up of potentially harmful code redundancy at worst.

4.2 A role-based architecture for Direct Combination

By contrast, a role-based DC architecture [8,9,10] has very good properties for highly distributed pervasive environments, and is well suited to coping with situations of rapid change. Role-based approaches in general are the subject of a great deal of research because of their great flexibility and extensibility [9,10]. Explaining the full details of the particular role-based approach we are using is beyond the scope of this paper, but we will briefly explain the key features of our role-based architecture, and then summarize the implications and advantages for supporting DC in pervasive environments. See [8] for a more detailed account. Those not interested in architectural technicalities may prefer to skip directly to section 4.2.1, which describes their practical implications.

- Role-based analysis [10] does not classify objects immutably. It focuses on representing the dynamically varying roles that objects may play. An object may typically support many roles simultaneously.
- Each role typically encapsulates a very narrow set of actions, or even a single action. So for example an electronic document might have the roles of being *renderable,* associated with the action *renderedBy*.
- Actions typically relate an object to one or more other objects playing complementary roles, e.g. *renderer*.
- A role may aggregate and modify other roles.
- Objects need not be defined by their class, but can be defined by aggregating roles (such as *renderable*, *authorable, virtual object* etc). Each role acts something like a small sliver of a conventional abstract class. By selecting and mixing roles, it is possible to describe new combinations of functionality without having to refactor classes.
- Crucially, in the particular system that we use to provide *direct support for role based programming*, known as UC [11], objects can *reflectively access all parts of all of their roles at run time*, even where these parts might clash if they were represented using multiple inheritance.
- This property allows potentially conflicting or contradictory roles to be treated very flexibly. The programmer may create new roles to manage the contention in line with any desired policy, with complete run-time flexibility.
- Direct support for role-based programming in this way means that most objects can be fully defined merely by specifying a list of roles (to specify its behaviour) and by plumbing in any relevant state.

Having outlined the key distinguishing features of our particular role-based approach, let us now consider how it applies to support the DC framework in pervasive environments, so that we can identify the implications and advantages.

• In order to define how two entities in a pervasive environment interact, it is not necessary to explicitly catalogue what reasonable interactions those two *entity types* should have. All that is necessary is for an analyst to consider the interactions possible for each individual *role*. This is simpler to analyse and less laborious to code.

• Once the interactions of roles have been identified, the interactions that an entity can enter into will follow automatically from its constituent roles. To analyse an entity for DC purposes, the domain analyst need only identify the roles that that particular entity can play. This is relatively quick and simple to analyse and code.

• For each set of roles that interact (e.g. *renderable and renderer*), the analyst must specify the following: a relevant operation (e.g. *render*), and a human-readable parameterised description of the operation.

• For easy interoperability, the operation should ideally be expressed in some universal protocol such as the future V2 standard [7].

• Computing the possible interactions between two entities at run time is computationally simple and cheap. For each role in turn, the corresponding role combinant is sought in the other entity. Every time a reacting pair is found, the descriptions of the corresponding operations are collected. These are then filtered to avoid duplication, and the options presented to the user.

4.2.1 Some advantages of role-based approach for applying DC to pervasive environments

Firstly, a role-based approach makes it relatively easy to *analyse* and *represent* the changing resources found in a pervasive environment, as outlined above. Resources can be represented simply by listing the roles that they play, and by plumbing in any necessary state. New roles can be identified by domain analysts and implemented as needed. Secondly, because of the same mechanism, a role-based approach is highly suitable for *distributed* computation, since in order to fully specify and reason about the functionality of an entity, it is only necessary to have the relevant list of roles. A third, perhaps unexpected advantage is the strong support given by this approach to perspective-based computing [8], which allows users to leverage the power of Direct Combination by making use of *viewpoints*, as explained in the next section.

5 Supporting Diverse Viewpoints

When designing Direct Combination systems, there is clearly flexibility in deciding precisely what set of operations should apply to what collection of selected objects. This flexibility, far from being a drawback, can be positively exploited as a major advantage. One very useful facility to be able to offer users and designers is the support of diverse *viewpoints*. This allows users to see the environment and the operations it offers from a variety of perspectives. Differing viewpoints are commonplace among, for example, different social, national and professional groups, and among those playing different temporary work or leisure roles. For example a fire-fighter, a policeman, a paramedic, an office worker, a tourist, and a child might identify the same object in different ways at different times, depending on the current role(s) they were playing: they might view the object as a potential fire-risk, a possible security breach, a health risk, a tourist attraction, an information source, a toy, etc. The notion of viewpoints can be very useful to the individual user, since it provides a convenient way of filtering and focusing options in very rich environments. For example:

Scenario 2.1

"How do I {reprogram this central heating system in the spare room, switch off the power to this wing of the house, switch off the water to the second floor}? Better set the Readers Digest Home Maintenance view on the wand."

Scenario 2.2

" How do I, a traveller, {buy a ticket at Ulan Bator railway station, change money, find accomodation}? Better set the Rough Guide view on my wand."

Scenario 2.3

"How do I get this door open? I wish I had a security guard's wand (and the biometric profile to make it work)."

For the support of diverse viewpoints to be possible, there are three prerequisites. Firstly, scaleable architectural support must be available. Secondly from an HCI perspective, viewpoints must be simple to manipulate and modify. Thirdly, we need to show that it is worth someone's while to maintain and represent the necessary substrate of information. We will begin with the architectural support. In an inheritance-based system, support for viewpoints can be provided using the mechanism of aspects [10]. However, the role-base architecture outlined above offers particularly good support for perspectives. In outline, a perspective is represented simply as a collection of roles. When viewing an entity from a given perspective, the reflective run-time properties of the architecture allow the

entity to behave in terms of the relevant roles alone. Even potentially conflicting purposes and functions may be separately elicited from a single object. This mechanism is well adapted for supporting viewpoints. As regards the user interface, the user need only specify a named viewpoint to filter, focus or afford interactions as desired (e.g. tourism, consumer association, electricians' union etc). Given the very simple composable nature of the mechanism, multiple viewpoints may be combined, or individuals could create their own viewpoints for specialised activities or roles. It should be clear that some viewpoints might be available only to those with the right professional credentials, passwords, paid up subscriptions or biometric data - in other words the system is amenable to a range of security models and business models. There would be a lot of scope for commercial and other organisations to use the system to leverage their specialised databases and services, making it worthwhile for useful viewpoints to be professionally defined and maintained.

6 Other Issues

The basic interaction styles of DC can easily accommodate many issues [13] such as resource discovery, feedback, monitoring of tasks in progress, cancelling, remote interactions, but these would take us beyond the scope of this paper. For a comparison of Direct Combination with a variety of other approaches to tangible and ubiquitous interaction, and a discussion of possible extensions to DC provide context-aware interactions, see [6].

7 Conclusions

In rich pervasive environments, end users will often dynamically create services of interest by causing *two or more devices or resources to interoperate together,* often under changing circumstances. In general, users find this kind of process hard to manage. Existing programming architectures make the situation difficult to address in a principled, scaleable way. Users find it particularly hard to manage such problems via devices with small, resource-poor user interfaces. We have demonstrated that a good theoretical basis for addressing an essential aspect of all of these problems is the theory of *Direct Combination*. Direct Combination allows the user to gesture at two or more objects (virtual or physical) to greatly constrain the search space for relevant commands. We have argued that a role-based DC architecture has particularly good properties for modelling rapidly changing and distributed pervasive environments where spontaneous interactions are commonplace. We have demonstrated how the concept of viewpoints can be used to filter and tailor the focus of operations with great simplicity and precision.

References

1. Newman, M.W., J.Z. Sedivy, W.K. Edwards, T. Smith, K. Marcelo, C.M. Neuwirth, J.I. Hong, and S. Izadi. (2002) Designing for Serendipity: Supporting End-User Configuration of Ubiquitous Computing Environments. In Proceedings of *Designing Interactive Systems (DIS2002)*, June 2, 2002. London, UK.
2. Banavar, G., Beck, J., Gluzberg, E., Munson, J., Sussman, J.B., Zukowski, D. (2000) Challenges: an application model for pervasive computing. In Proceedings of the sixth annual international conference on Mobile computing and networking MOBICOM 2000 pp 266-274. ACM Press.
3. Keith Edwards and Rebecca Grinter. (2001) At Home with Ubiquitous Computing: Seven Challenges, Ubiquitous Computing 2001, Atlanta, GA, September 2001.
4. Kristoffersent, S. and Ljungberg, F., (2000) Representing Modalities in Mobile Computing: A Model of IT-use in Mobile Settings. White Papers, Norwegian Computing Center, Oslo.
5. Holland, S. and Oppenheim, D. (1999) Direct Combination. In Proceedings of the ACM Conference on Human Factors and Computing Systems CHI 99, Editors: Marion Williams, Mark Altom, Kate Ehrlich, William Newman, pp262-269. ACM Press/Addison Wesley, New York, ISBN: 0201485591.
6. Holland, S., Morse, D.R., Gedenryd, H. (2002) Direct Combination: a New User Interaction Principle for Mobile and Ubiquitous HCI. In Proceedings Mobile HCI 2002, LNCS, Springer Verlag.
7. Zimmermann, G. Vanderheiden, G, Gilman, A. (2002) Prototype Implementations for a Universal Remote Console Specification. CHI 2002 Extended Abstracts , pp 510 - 511. ISBN:1-58113-454-1
8. Gedenryd, H., Holland,S. Morse, D.R. (2002) Meeting the software engineering challenges of interacting with dynamic and ad-hoc computing environments. Technical Report, Computing Dept, Open University, Milton Keynes, MK76AA, UK.
9. G. Gottlob, M. Schrefl, and B. Rock. (1996) Extending object-oriented systems with roles. ACM Transactions on Information Systems, 14(3):268-296, 1996.
10. Reenskaug, T, Wold P. and Lenhe O.A. (2001) "Working with Objects, The OOram Software Engineering Method". By Trygve Reenskaug, with Per Wold and Odd Arild Lehne. ISBN 1-884777-10-4
11. Gedenryd, H. (2002) Beyond Inheritance, Aspects and Roles: a Unified Scheme for Object and Program Composition. Technical Report, Department of Computing, The Open University, Milton Keynes, MK76AA, UK.
12. Weiser, M. “The Computer For the 21st-Century,” Scientific American, vol. 265, pp. 66-75, 1991.

13. Victoria Bellotti, Maribeth Back, W. Keith Edwards, Rebecca E. Grinter, Austin Henderson, Cristina Lopes (2002) Ubiquity: Making sense of sensing systems Proceedings of the SIGCHI conference on Human factors in computing systems: Changing our world, changing ourselves April 2002.