Reducing Configuration Overhead with Goal-oriented Programming

Justin Mazzola Paluska, Hubert Pham, Umar Saif, Chris Terman, and Steve Ward
{jmp,hubert,umar,cjt,ward}@mit.edu
MIT CSAIL
32 Vassar Street
Cambridge, MA 02139

***Abstract*— The rapid increase in the number and variety of consumer-level electronic devices without the corresponding development of device management technology has lead to a configuration nightmare. We propose to use goal-oriented programming over a substrate of network-portable objects to help reduce the amount of configuration users must do in order to have their applications use their devices efficiently. We detail an architecture and describe a prototype system using existing pervasive computing technology that plays music on the most appropriate devices without requiring user interaction and configuration.**

I. Introduction

Recent years have seen an explosion in the number and diversity of consumer-level electronic devices. Many of these devices work extremely well in one environment in a few scripted ways, but have the possibility of working in many other, unanticipated ways if users are willing to expend considerable effort configuring the devices and connecting them together. Unfortunately, this effort confounds even determined users with time and money to hire professional installers and technical support [1].

As a motivating example, suppose a user would like to play a video from her laptop computer. The computer normally plays video on its own screen and internal speakers, but the user enjoys the experience more on her television and home theater system. As such, every time the user wants to watch a video from the laptop, she must:

1) Connect the laptop's television output—if it has one—to one of television's inputs,
2) Connect the laptop's audio outputs to an adapter bought from RadioShack then that to the speakers' inputs,
3) Activate the inputs of the speakers and television,
4) Start the media player on the computer,
5) Instruct the laptop to render the media to its external outputs, and
6) Press "play" and hope that the television and computer agree on aspect ratios and other miscellany.

If the user would rather move the laptop to another room and play on the laptop's built-in components, she must reconfigure the laptop once again.

We believe that these configuration hassles can be reduced most of the time to "just play".

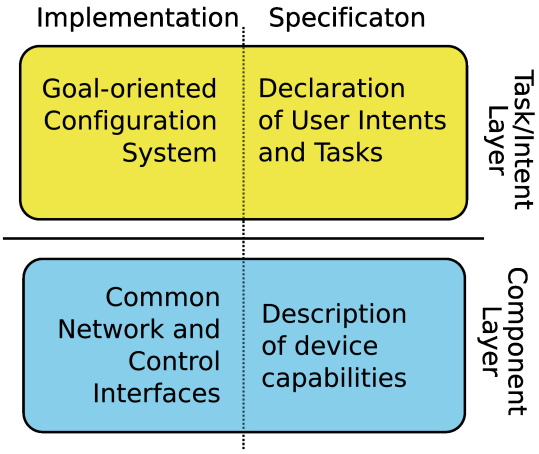

Fig. 1. Software architecture for "Just Play"

II. The "Just Play" Proposal

The steps above show that configuration is difficult because devices have different and often incompatible physical interfaces; because devices cannot be controlled in a standardized way; and because software is application-centric rather than task-centric. As a result, the user must explicitly mediate between each pair of devices that she wishes to communicate and run software to enable that communication. Instead, we would like a dynamically configured device environment where users tell their devices what high-level tasks they would like to accomplish and the system figures out the low-level details of those tasks. In such a system, the user tells the system to "Play Video" and the system responds by searching out for a way to "Play Video" given the resources at hand.

In order to build this world we need a way to capture what the user wants, a way to translate user intent into machine-implementable tasks, and a way of implementing the tasks with real software and hardware. We propose the following four part, two-layer architecture to enable users to better use their devices by enabling automatic configuration:

1) **Machine-readable User Intents** to allow a user's abstract desire to be read by the automatic configuration

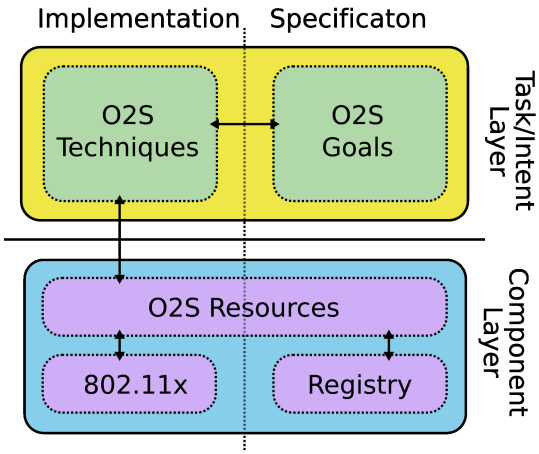

Fig. 2. Software components used for the "Just Play" prototype.

system.

2) **A Goal-oriented Configuration System** to match user's intents with recipes, functions, and devices that can achieve those intents.
3) **A Common Wireless Interface and Control Mechanism** to eliminate physical connection mismatches and allow replacement of devices by other devices over time.
4) **Low-level Descriptions of Device Capabilities** so that devices may be used in ways beyond their single scripted use. For example, a television is not just a television, but a display device, NTSC or PAL tuner, audio output device, and a remote control receiver, all of which might be useful in implementing user tasks.

The first two architectural points form a layer that captures and satisfies user tasks. The last two architecture points form a layer that implements ways of satisfying tasks given devices of certain capabilities. Figure 1 illustrates our architecture. While there is some architectural fluidity within the layers, we maintain a strong abstraction between the two layers. This enables the two layers to evolve independently and be to replaced as new technology develops.

III. Initial Proof-of-Concept

We are currently developing a set of prototype devices and a prototype programming environment in which we can explore the "Just Play" architecture. Figure 2 illustrates our approach and choice of components. The user of our system speaks to a "Voice Shell" on a nearby computer that performs voice recognition on her utterance and translates it into a (currently) small universe of intents. When the user is done listening to the music, she simply clicks a button or tells the system to stop.

For example, the utterance "Play me some jazz" gets translated into a high level intent of `PlayMusic(type=jazz)`. The system finds a jazz music stream and a music output device from the devices available and accessible to the user on

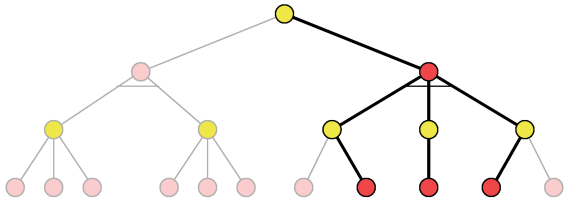

Fig. 3. A sample Plan Tree. Goal nodes are yellow. Technique nodes are red and have their dependencies tied together with a cross-bar, since all dependencies must be satisfied before the Technique can proceed. Dark lines and darkly colored circles represent the nodes currently chosen by the Planner.

the network. If a new device becomes available—for example, the user plugs in a speaker she just purchased—the system evaluates if the new device is better than what is currently being used and, if necessary, switches to the new device. Similarly, if a device suddenly becomes unavailable, e.g., due to power loss, the system plans around it. When the user signals that the system shouldn't play music anymore, the system fades out the audio and disconnects the music stream.

In essence, the system maintains a formal record of the user's current intents (goals) and doggedly pursues them so long as they remain active, using the best resources it can find. It adapts to newly-available resources without explicit reconfiguration, eliminating configuration overhead in a variety of common system extension scenarios.

A. Implementation Details

We extend elements of MIT's O_2S [2] pervasive programming system for our software infrastructure. In particular, the prototype uses an extended version of the O_2S Goals and Techniques programming system for the Task/Intent layer. Goal-oriented programming [3] provides a way of writing applications so that the algorithms, devices, and resources used can be evaluated and exchanged for ones with similar functionality as needed. For the Component layer, we use O_2S Resource network-portable objects [4] over 802.11 wireless networks. In particular, the O_2S resources framework provides us with a simple discovery system and a common interface through which to access devices as objects.

Hardware-wise, our prototype "Just Play" environment uses a Mac Mini, a standard laptop, and a speaker modified with an 802.11b wireless interface and MP3-to-analog decoding hardware. The Mac Mini acts as a trusted wireless and computation hub that can monitor the state of the network and discover devices on the user's network. The laptop acts as a UI, a source of music, an audio input device, and an audio output device. Finally, the speaker simply provides audio output. However, unlike conventional speakers, we modified ours so that when it is plugged into power, it discovers the O_2S Registry on the Mac Mini and informs it of its capabilities as an audio output device.

B. Goal-oriented Programming

Goal-oriented programming offers two primary abstractions. *Goals* act as a specification abstraction. Goals are satisfied by

Techniques, a mixture of declarative statements and arbitrary code.

Unlike traditional programming models, Goals are not bound to any particular Technique until runtime. The binding is mutable and may change as better devices come up or the context in which a particular binding choice was made is no longer valid. Thus, Goals provide a natural way of expressing the points in an application that can be swapped out and replaced as needed. The binding of Goals to Techniques is handled by a Planning engine that cooperates with Techniques to find the best way to satisfy the user's top-level Goals.

A Technique may be Goal-oriented by declaring subgoals that the Planner, in time, recursively satisfies. The "return value" of a subgoal declaration is a *Solution* object that the Technique can use to access properties of the subgoal, i.e., the sub-Technique the Planner chose to for the subgoal. This information, in turn, is used by the Technique to set properties of its own solution. Solutions can contain arbitrary properties but must contain a "satisfaction" property that measures how well the Technique is satisfying its own Goal.

The O_2S Planner requires that all of a Technique's subgoals be satisfied before that Technique can proceed. The blocked Technique, however, does not block other Techniques from proceeding as long as their own subgoals are satisfied. Goals, on the other hand, can be satisfied by any one of a variety of Techniques. As such, Goals and Techniques form an AND/OR tree. Our "planning process" is simply heuristic search from the root to the leaves of each plan tree for the best Techniques for each Goal as measured by "satisfaction".

Figure 3 illustrates a sample *plan tree* for an instantiation of the `PlayMusic(type=jazz)` Goal. Note that the plan tree is just a dependence tree. The code in Techniques determines the final structure of the application.

C. Techniques

Goals are just specification—the real work of connecting devices together is done by Techniques. Figure 4 shows a sample Technique to `PlayMusic via MP3s`. The first line declares the Goal that the Technique satisfies; the `via` statement just names the Technique to ease debugging. The rest of the Technique is divided into stages:

```
to PlayMusic(type):
  via MP3s:
  subgoals:
    source = MusicStream(type,
                         format="MP3")
    control = VolumeControl()
    sink = AudioSink(format="MP3")
  eval:
    satisfaction = (source.satisfaction +
                    sink.satisfaction) / 2
  exec:
    connect(source, sink)
    control.set_speaker(sink)
  update sink from old:
    disconnect(source, old)
    connect(source, sink)
```

Fig. 4. Sample `PlayMusic` Technique

- A `subgoals` stage that declares the subgoals of the Technique.
- An `eval` stage that investigates the results of the subgoals and synthesizes how well the Technique can satisfy its own goal given the Planner's choice of subgoal implementations.
- An `exec` stage that contains the code to run if the Planner chooses the Technique for execution.
- An `update` stage that contains code to swap out the `sink` subgoal for another one if necessary.

Technique code is not run all at once. Instead, the Planner runs and re-runs stages as necessary to build a plan tree, evaluate what Techniques form the best plan, execute the best, and the monitor the chosen Techniques.

D. "Just Play" and Goal-oriented Programming

We use Goals to capture user and programmer intents. Since Goals are explicit adaptation points, we can use the indirection Goals provide to automatically choose and configure devices and service for users as long as choices meet the Goal's specification. In particular, we do this by writing Techniques that describe common configuration patterns; the Planner is left to evaluate which is best at any given time and run them.

In the "Just Play" prototype, we have roughly two kinds of Techniques. The first kind are Techniques that function much like device drivers in traditional operating systems. These simply gather network representations of each device and copy their properties to Solution objects. The other kind of Techniques are "algorithmic" Techniques that embody the logic of how components are connected together. The Technique of Figure 4 is a typical algorithmic Technique, as it assumes its subgoals are satisfied and simply provides a recipe for connecting them together. In order to give the Planner the most choices, our prototype Techniques use subgoals whenever possible.

The plan tree of Figure 5 is a snapshot of the "Just Play" system during runtime. At the highest level is the `PlayMusic` Goal that the user invokes when she wants to listen to music. The `PlayMusic` goal is satisfied by the `via MP3s` Technique. Lower-level Techniques satisfy the subgoals of the `via MP3s` Technique; the recursion terminates at Techniques that represent each available device in the system.

Our system is additive and extensible: As new ways of playing music or new devices come along, we only need to add new Techniques to the Planner for those devices or methods, giving it more choices when satisfying the user's `PlayMusic` Goal.

E. The Component Layer and Portability

The O_2S Resources system provides a network portable object interface as well as device discovery and process health monitoring. Low-level Techniques that must find devices and services interface directly with the Component layer. For example, as part of it's health monitoring service, the Component layer sends asynchronous `DEVICE UP` and `DEVICE DOWN` events. Low-level Techniques convert these into satisfaction

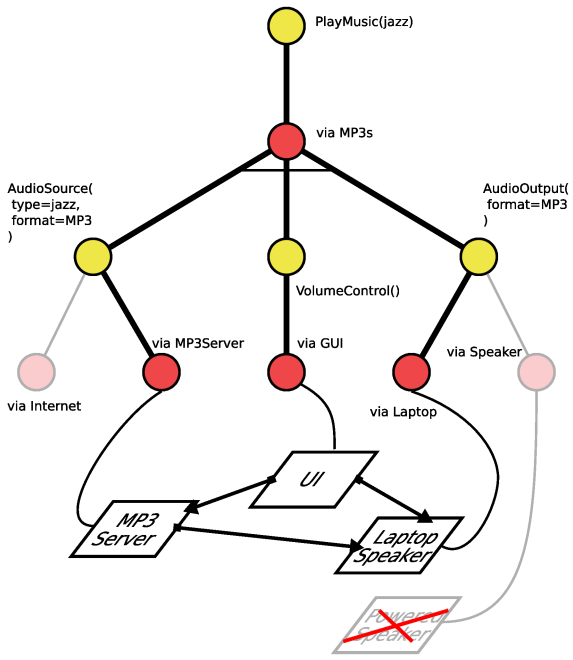

Fig. 5. Snapshot of the "Just Play" plan tree during runtime when the speaker is unavailable. Note that the Plan Tree reflects module dependencies and is independent of the final application structure.

values in their solution objects, e.g., 1 for a working device, and 0 for a non-working one. This way, the rest of the system can be used even if the implementation of the component layer changes.

While we chose to work with familiar and integrated components of O_2S, our architecture does not require the use of O_2S. For example, it is possible to swap out O_2S resources with a combination of Sun's RMI [5] and the IETF's Service Location Protocol (SLP) [6]. This would require reconfiguring the lowest-level Techniques that deal directly with devices, but not necessarily all Techniques. We believe that algorithmic Techniques would need few changes.

IV. Related and Future Work

The "Just Play" framework draws ideas from many pervasive computing systems. UIUC's Gaia [7] and CMU's Aura [8] both add a level of indirection between traditional applications and I/O devices. Similarly, systems like Ninja Paths [9] and SoNS [10] allow stream-based applications to transparently change endpoints. Olympus [11] extends Gaia with portable scripts that tie together resources within an intelligent room. Techniques are similar, but are not tied to a specific class of physical environments: the provide a generic programming model that we believe is useful for many kinds of pervasive environments. We, like PCOM [12], take the more radical approach to system decomposition than these systems by breaking apart previously atomic applications. In our case, we divide into Techniques.

UPnP [13], Bonjour [14], and SLP [6] each provide discovery methods. They are best suited as replacements for discovery part of the Component layer—configuration must be built on top of the substrates they provide. Closer to our work is ISI's Pegasus [15], which applies AI planning algorithms to generating scientific workflows on the Grid based on discovered resources. We differ in focus, however, as Pegasus is based on moving files and programs around for execution, not for device configuration.

There are two areas in which we need to extend our prototype. First, we need to find a way to secure devices on the device network without configuration hassles. Whereas a laptop and speaker wired together are authenticated to each other by the wire between them, wireless networks offer no such authentication. This problem is compounded if users want to be able to share their device networks or use devices—such as projectors—that are a part of a shared infrastructure owned by no one user.

Second, we need to determine how to best give non-programmers a choice among equally good ways of satisfying a high-level goal. Programmers can write their own Techniques to do so, but we cannot expect most users to do this. A simple interface would be to choose a single plan and switch among them if the user rejects it. Alternatively, another interface lets users explicitly tell the system to use a device in a particular way, perhaps using speech recognition or gesture recognition.

References

[1] A. Marcus, "The out-of-box home experience: remote from reality," *interactions*, vol. 12, no. 3, pp. 54–56, 2005.
[2] U. Saif, H. Pham, J. Mazzola Paluska, J. Waterman, C. Terman, and S. Ward, "A case for goal-oriented programming semantics," in *UbiSys 2003*, 2003.
[3] J. Mazzola Paluska, "Automatic implementation generation for pervasive applications," M.Eng Thesis, Massachusetts Institute of Technology, June 2004.
[4] H. Pham, "A distributed object framework for pervasive computing applications," Master's thesis, Massachusetts Institute of Technology, 2005.
[5] Sun Microsystems, "Java remove method invocation," http://java.sun.com/rmi, 1994.
[6] IETF, "Service location protocol, version 2," RFC 2608, June 1999.
[7] M. Román, C. Hess, R. Cerqueria, A. Ranganathan, R. H. Campbell, and K. Nahrstedt, "A middleware infrastructure for active spaces," *IEEE Pervasive Computing*, pp. 74–83, October-December 2002.
[8] D. Garlan, D. P. Siewiorek, A. Smailagic, and P. Steenkiste, "Project aura: Toward distraction-free pervasive computing," *IEEE Pervasive Computing*, pp. 22–31, April-June 2002.
[9] S. D. Gribble, M. Welsh, J. R. von Behren, E. A. Brewer, D. E. Culler, N. Borisov, S. E. Czerwinski, R. Gummadi, J. R. Hill, A. D. Joseph, R. H. Katz, Z. M. Mao, S. Ross, and B. Y. Zhao, "The ninja architecture for robust internet-scale systems and services," *Computer Networks*, vol. 35, no. 4, pp. 473–497, 2001.
[10] U. Saif and J. Mazzola Paluska, "Service-oriented network sockets," in *MobiSys 2003*, 2003.
[11] A. Ranganathan, S. Chetan, J. Al-Muhtadi, R. H. Campbell, and M. D. Mickunas, "Olympus: A high-level programming model for pervasive computing environments," in *PerCom 2005*, March 2005, pp. 8–12.
[12] C. Becker, M. Handte, G. Schiele, and J. Rothermel, "Pcom — a component system for pervasive computing," in *PerCom 2004*, March 2004, pp. 67–76.
[13] Universal Plug and Play Forum, "Universal plug and play (upnp)," http://www.upnp.org/.
[14] Apple Computer, "Bonjour," http://images.apple.com/macosx/pdf/MacOSX_Bonjour_TB.pdf.
[15] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny, "Pegasus: Mapping scientific workflows onto the grid," *LNCS*, vol. 3165, pp. 11–20, 2004.