# A Case for Goal-oriented Programming Semantics

Umar Saif, Hubert Pham, Justin Mazzola Paluska, Jason Waterman, Chris Terman, Steve Ward
{umar, hubert, jmp, jwaterman, cjt, ward}@mit.edu

**Abstract.** *Contemporary pervasive computing environments demand mechanism for coherently addressing high-level user needs despite changing availability of resources. We propose the formalization of* **goals** *as the semantic basis for this mechanism, and sketch a system architecture that separates policy-rich goals-level planning code from a policy-neutral component assembly model.*

## I. INTRODUCTION

Pervasive computing systems immerse their users in a triad of sensors, invisible servers and mobile devices that work together to satisfy user requirements according to the facilities available in her environment. Such a system must be self-managing: it must be able to continuously monitor changes in user locations and needs, respond both to component failures and newly available devices, and maintain continuity of service as the set of available resources change.

As a simple example of such a pervasive computing environment, consider a user involved in a video-conference with a colleague as she wanders about a well-equipped campus. As she moves from one room to another, her video may switch from the small LCD display of her handheld to a wall-mounted plasma screen as the latter comes into view; networking technologies might shift between 802.11b and CDMA depending on resource availability, and video may degrade or disappear altogether as communication bandwidth warrants.

Such adaptation in a pervasive computing environment entails more than traditional load-balancing or resource-management at a single host; it necessitates a certain degree of planning involving continuous reevaluation of available alternatives, as well as heuristic compromises to best address the user requirement using imperfect resources in the changing environment of the user.

Conventional techniques for constructing distributed applications, in which a top-level function is decomposed into statically-partitioned sub-functions, each affixed to a particular API, makes such adaptation exceedingly difficult to program. Adaptation in a pervasive computing environment requires planning at a macro-level, possibly involving a wholesale re-structuring of the application. If there is a change in available resources or user priorities, it is often insufficient simply to reconsider how to implement the function specified at each API: it is necessary to reconsider the reason that API was selected, and whether an alternative function and API has now become more appropriate.

A more promising approach is to have the user express their requirements as an abstract high-level goal, and then let the system automatically satisfy this goal by assembling, on-the-fly, an implementation that utilizes the resources currently available to the user. The high degree of dynamism in the environment requires that the resolution of a goal not be a static one time process. Instead, the system must be able to continuously monitor the environment so that it may respond opportunistically to changes in connectiv-

ity and available devices, assembling new implementations to sustain the high-level goal in the changing conditions.

While a growing number of projects[1][2][3] discussed in section 4 show how conventional distributed computing paradigms may be extended to handle the heterogeneity and dynamism of a pervasive computing environment, we believe that a new set of abstractions would greatly reduce the complexity of creating applications for this environment.

To this end, we are building an experimental system, O2S, to explore two architectural principles:

- The formalization of goals as an explicit semantic construct to express abstract high-level intentions. A goal takes the form of a generic procedure call and represents the stimulus for a planning process which can be invoked repeatedly to produce successive implementations as available resources evolve.
- The stratification of application-level software into two distinct layers: a planning layer that embodies the mechanism for assembling and adapting implementations to be executed by the computation layer. The computation layer provides reasonably general mechanisms for constructing and monitoring a network of generic modules operating in parallel on different hosts. The two layers are an attempt to tease apart software for complex, adaptive systems into cognitive and reflexive components: the former focusing on application-specific policies, and the latter on efficient deployment and monitoring of commodity computation.

By following these two architectural principles, O2S offers a general-purpose architectural framework for engineering goal-oriented adaptive systems. The approach of separating the adaptation mechanism from individual code modules leads to the following useful properties: 1) adaptation could be performed at a macro-level, allowing wholesale restructuring of the application-code in response to changes in the system, 2) policies, algorithms and mechanisms of adaptation can be engineered and evolved independently of any particular application.

## II. GOALS AND PLANNING

In the planning layer of O2S, *goals* are formalized as a language construct and used to guide the automatic construction of a component-based system. Syntactically, goals are similar to generic procedure calls: they involve a named generic service (the goal name) as well as an arbitrary number of typed parameters. Thus, **TeleConference(Alice, Bob)** is a high-level goal whose satisfaction requires a teleconference link between **Alice** and **Bob**, each a parameter of type **Person**.

Unlike procedure calls, however, goals are disembodied from any block of code to be invoked during their execution. Rather, the system approaches the resolution of a goal by searching for one or more *techniques*, each of which constitutes a recipe for resolving a class of goals. Each technique specifies a pattern to be matched against a target goal, optional sub-goals that must be satisfied for that technique to proceed, and code to be run in order to cause the target goal to be satisfied once the specified subgoals have been achieved. Satisfaction of a top-level goal thus involves, at least conceptually, (a) the enumeration of applicable techniques; (b) evaluation of each of the candidate techniques; (c) the heuristic selection of one technique for implementation; and (d) implementation of the chosen plan. Any of these steps may fail, possibly resulting in the system's failure to satisfy the specified goal.

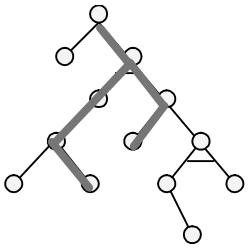

**Figure 1:** O2S Goal Tree

Since techniques can require subgoals to be satisfied, step (b) in the above sequence may involve the recursive evaluation of an entire tree of goals and subgoals. In practice, O2S builds a *goal tree* (shown in figure 1) representing a (perhaps partial) universe of candidate choices, and heuristically selects a path through this tree based on an estimate of the most acceptable implementation choice (as outlined in more detail subsequently). The resulting goal tree, however, is saved after the implementation choices are made; it serves as a record of the assumptions and logic leading to each decision made in the heuristic selection, allowing each decision to be reconsidered should new information become available.

**Goal Resolution**

The planning layer of O2S is implemented as a distributed network of *contexts*, each of which contains a local repository of techniques, often customized to the preferences of some logical entity in the system. For instance, an individual will typically maintain a personal context whose techniques reflect her preferences (e.g., a GetMyAttention goal is satisfied by a local technique that causes her cell phone to vibrate); however, the personal context may defer to a context maintained by her employer to specify default techniques for the satisfaction of many goals. The principal function performed by a context is the resolution of a goal instance provided as a parameter in a **satisfy** request. As noted above, the approach to goal resolution is expected to evolve; indeed, the O2S architecture is intended to provide a framework for that evolution. In this section, we describe the approach taken by our early prototypes.

Both contexts and techniques are implemented as object instances and their external interfaces take the form of method-call APIs. The structure of the goal resolution take the form of a choreographed interaction between the context and a selection of the techniques it houses, roughly as diagrammed in Figure [2].

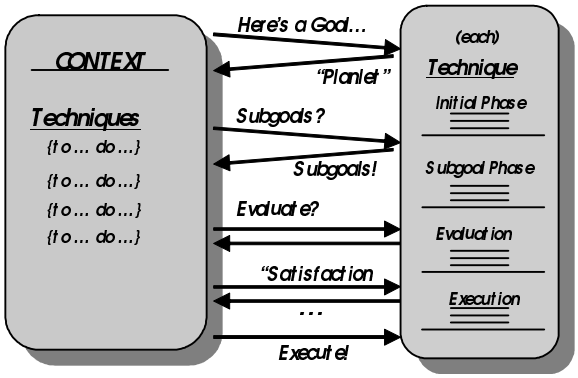


**Figure 2:** Context/Technique negotiation

A target top-level goal (typically derived from a user command, either typed or spoken) is first matched against the local store of techniques. Each technique is matched against the target goal; if the match succeeds, the technique is retained for evaluation. If

no matches are found, the satisfy request reports failure (but the failure may be recoverable, as discussed below).

Each matching technique returns an instance of a *Planlet* object used to cache the state of subsequent negotiation relating to that technique's suitability to satisfy the specified goal. The set of planlets are collected by the context into a *plan*, recording the state of the search which will eventually lead to the heuristic selection of one planlet as the path of choice. The plan, and its child planlet instances, constitute one layer of the unfolding goal tree devoted to choosing an implementation strategy for the incoming goal instance.

Additional layers of the goal tree are built by interrogating each planlet for subgoals, and repeating the above construction of a plan (with subordinate planlets) for each. The resulting tree is structured as alternating layers of *plan* and *planlet* nodes, which represent conceptually disjunctions and conjunctions of their inferiors. Each plan can succeed if one of its inferiors – the use of some suitable technique – leads to success; each planlet can succeed only if all of its inferiors, each corresponding to a requisite subgoal, is successful.

The next step in the goal resolution process is the heuristic selection of a subtree that represents, by some criterion, the best implementation choice given currently available information. To make this determination, our current approach involves assignment of a scalar satisfaction metric to each node of the tree, and the local selection at each plan of the inferior planlet with maximal satisfaction. This primitive approach involves the encapsulation of all value judgments pertaining to a choice, including both its cost and the desirability of its outcome, onto a single dimension. Currently the metric is computed are by code within each technique, which is asked by the context to estimate the satisfaction of its outcome based on goal parameters and the results (including satisfaction) of the resolution of each of its subgoals. Standards for such value judgments are thus embedded in techniques, and the choices made by a context in the realization of incoming goals may be heavily colored by the selection of techniques it comprises.

The evaluation of a planlet is required to be idempotent, and may be done repeatedly during its lifetime. This allows flexibility in the evaluation algorithm used by a context, as well as providing the ability for the subsequent re-evaluation of any subtree should some incoming event suggest that its result might change. The plan, and the substructure that constitutes its goal tree, persists while the chosen implementation is active. Various external events, such as an error condition or change in the location of a principal, may be used to trigger re-evaluation of all or part of the tree; this may, in turn, result in a (partial) reimplementation of the solution.

It is worth mentioning that the specific heuristics of our current planning system, including the use of scalar satisfaction metric, are early steps in an ongoing research agenda. We view our architecture as defined by its network APIs rather than, say, the body of code that implements key heuristics such as goal resolution. We anticipate that alternative implementations of contexts (incorporating different approaches to planning) might interoperate simultaneously within a continuously-running O2S environment, allowing incremental evolution of the planning subsystem. Among many other possibilities, we are exploring the use of current SAT-based planning techniques [4] as an alternative to the PROLOG-style planning described here.

## III. COMPONENT ASSEMBLY

A target goal is satisfied at conclusion of the iteration of the planning process by dynamically assembling a set of generic components (called *pebbles*) to implement the high-level function encoded by the goal. A pebble is a lightweight, policy-neutral distributed component that conforms to a standardized API. Pebbles typically implement a single function and are designed to be standalone components with well-defined, explicit ports of communication with other components. Keeping the pebbles focussed on a single operation makes it easy to reuse them in many different applications. For example, typical pebbles in our prototype system include a voice-recognition pebble, which takes an audio-stream on an input port and produces corresponding recognized text on its output port, an audio-source pebble which reads /dev/dsp and forwards that on its output port, an audio-sink pebble which accepts an audio-stream on its input port and writes to /dev/dsp of the host machine.

***Component ("Pebbles") API***

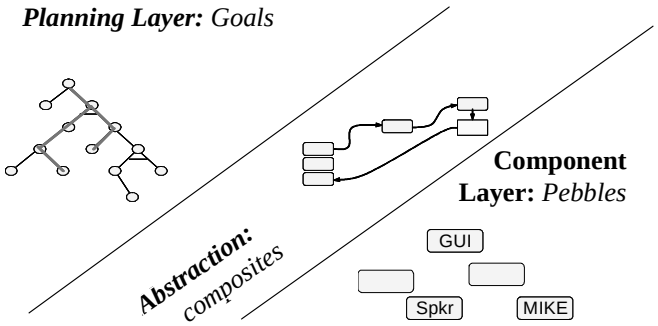


Figure 3: Layers of abstraction in O2S

The API presented by our component model is designed to hide certain complexities of a highly parallel, distributed system of interconnected components under the veneer of a simple, sequential, and localized system. Although our design is motivated by the implementation needs of our goal-oriented planning layer, the pebbles-based computation layer is designed to be usable by (and of potential interest to) those following more conventional approaches to distributed application design.

The basic API provides a mechanism for instantiating a collection of pebbles on various hosts, interconnecting them into a network, and monitoring the operation of the resulting *composite* via a stream of high-level events generated by the pebbles and connections. Events are used to report component failures, user inputs, or various pebble-specific notifications.

The health of devices hosting pebbles (and the communication paths between them) is transparently monitored by keep-alive connections; pebble state updates and debugging output are collected, filtered, and serialized for presentation to the caller; and disconnected or abandoned component processes are automatically garbage-collected. The intent is to minimize the tedium of developing and maintaining distributed applications, allowing a high-level application to focus on a simple sequential model for its "cognitive" deliberations, while the compute-intensive "reflexive" components are managed largely automatically (refer to figure 3).

This approach is similar to architectural models proposed as a basis for evolvable and self-repairing software [5]. The pebbles system exports a simple architectural model in which the application is represented as a graph of connected components. This representation is also adopted by a number of architecture description languages, e.g Acme [6]. By the same token as a software-architectural model provides the generic abstraction for applying analytical models to the overall architecture of a distributed application, the pebbles' composite serves as the generic abstraction for planning the assembly and adaptation of code modules to satisfy user goals.

**Connections**

Pebbles are modeled as stream-processing objects [9], and communicate using standardized communication end-points. An application connects two pebbles by requesting the system to establish a communication channel that joins their appropriate service-access-points, dubbed as *connectors* [10]. A connector defines a unidirectional interface for message passing, analogous to a UNIX one-way socket. All connectors are derived from a common super-class and hence provide a standardized, well-known interface for sending and receiving messages. The use of a standardized, low-level communication interface avoids the interface compatibility issues associated with conventional RPC-based distributed components, making it easy to join arbitrary pebbles. Where *connectors* are akin to UNIX sockets as a well-known message passing interface, the communication model supported by connectors enforces a strict distinction between policy and mechanism: a pebble only declares a connector when defining an external service access-point, while the actual connections between pebbles are established by the planning layer to compose an application. This allows pebbles to offer a service without dictating their role in the overall distributed computation, permitting a wholesale restructing of the application by reconnecting the pebble connections.

It is noteworthy that a connector declaration *only* defines a *service access point* and dictates neither the pebble to be connected to the connector nor the attributes or the protocol to be used for the transport of messages sent via the connection. These architectural principles, often referred to as "laws of blind communication" in the software engineering literature [5], ensure that a pebble can be used to satisfy a variety of high-level goals, that a pebble is independent of the semantics of the connectors to which it is attached, and that a pebble composite can be re-wired on the fly in response to changes in user priorities or needs.

## IV. RELATED WORK

A number of recent architectural approaches to pervasive computing share elements with our work. Several of the most relevant are discussed below.

MIT's Intentional Naming System (INS) [3] addresses these challenges at the stage of discovering and routing of network messages. Applications in INS express their *intent* as a set of properties required in a suitable resource and the INS overlay locates and routes *intentional datagrams* (network messages tagged with user intent) to the most appropriate resource in the environment of the application. This approach of pushing intent-driven adaptation down to the primitives of network communication enables conventionally-engineered applications to opportunistically access resources in a dynamically changing pervasive environment, and provides a runtime alternative to the top-level goals of O2S. It

does not, however, automate the internal structuring of an implementation, e.g. by the decomposition of goals into lower-level subgoals.

CMU's Aura [2] distributed computing environment defines an high-level abstraction, termed a *task*, layered atop individual applications. This task layer, called *prism*, provides a placeholder to capture user intent, and employs various resource monitoring artifacts in the Aura system to monitor and adapt underlying applications to opportunistically carry out the high-level task. Aura's architecture is focused on adaptation and migration of conventionally-engineered applications rather than on automatic assembly of distributed components.

The Gaia computation environment from UIUC [7] defines a programming environment based on the Model-View-Controller abstraction. Using this abstraction, applications in Gaia are partitioned into four parts; a model to encode the logic of the application, a view to expose the model's state, a controller to map events in the environment of the application as input messages to the model, and a coordinator responsible for storing the bindings of different components in the application's model as well as mechanisms to access and alter these bindings. While this conceptual framework reflects a philosophy similar to that of O2S in which the application-logic is separated from individual components, the Gaia architecture defines a basis for modeling user-driven context-aware applications, rather than automatic composition and runtime adaptation of distributed applications.

The Ninja Paths architecture [11] from University of California Berkeley takes a similar approach to O2S for automatically assembling a stream-conversion application. Given a source and a destination stream format, the NINJA Paths architecture strings together a path of stream-processing elements, dubbed as operators, which converts the stream from one format to another. Operators describe their properties, including the ingress and egress stream formats, in an XML file which is used by the NINJA Paths creator to discover the appropriate operators. Operators communicate using well-defined ports of communication which can be dynamically rewired in response to operator failures.

The Pebbles' composite API is reflective of the architecture-based software-engineering models, in which the architectural model of the software is maintained at runtime and used as a basis for software evolution and verification. Oriezy et al. [5] present a data flow based approach to compose and adapt a network of distributed components, called Weaves. Much like a pebbles composite, a weave is constructed by connecting a network of tool fragments using transport services. However, the Weaves system does not provide any explicit support for representing change policies to compose and adapt a weave. Weave designers use an interactive, graphical editor to visualize and directly reconfigure a weave during runtime.

Cheng et al. [6] argue for the suitability of software architectural models as a basis for analyzing and verifying self-adaptive applications in a pervasive computing environment. Much as O2S defines a layered architecture to separate the concerns of goal-resolution and planning from component assembly, Cheng et al. propose a dichotomy which separates analytical methods for evaluating the properties of the system's architectural design from individual code modules. Though similar to Pebbles in its architectural considerations, Cheng. et al. propose their system as a basis for analyzing and verifying the overall structure of an distributed application against programmer-defined constraints rather than goal-driven planning and assembly of distributed components.

## V. CONCLUSION

We propose *goals* as an explicit semantic construct for programing a pervasive computing environment. Goals encode high-level intentions, take the form of a generic subroutine call and initiate a planning process which may persist during a succession of implementation choices -- each potentially involving different design approaches, recompilation of code, and revised user-visible performance. Each iteration of the planning process satisfies the target goal by assembling a network of pebbles (modular components) and communication channels. The pebbles component system is based on an explicit distinction between mechanism and policy; individual pebbles offer a mechanism without dictating the policy specifying the role of the pebble in the overall logic of an application. Pebbles are intended for reuse in implementing a variety of applications. The API offered by the pebbles layer provides simple means by which the caller can instantiate a collection of pebbles on various hosts, interconnect them into a network and monitor the operation of the application via a stream of events generated by the pebbles and connections and adapt its structure in response to the changes in the system.

## VI. REFERENCES


[1] Robert Grimm *et al., Systems directions for pervasive computing.* Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII), pages 147-151, 2001.

[2] David Garlan, Dan Siewiorek, Asim Smailagic, and Peter Steenkiste. *Project Aura: Towards Distraction-Free Pervasive Computing.* IEEE Pervasive Computing, April-June 2002.

[3] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. *The design and implementation of an intentional naming system*. Symposium on Operating Systems Principles, Kiawah Island, USA, December 1999.

[4] Kautz, H., and Selman, B., *BLACKBOX: A New Approach to the Application of Theorem Proving to Problem Solving*, In Working notes of the Workshop on Planning as Combinatorial Search, AIPS-98 Pittsburg, PA, (1998) 58--60.

[5] Oriezy, P., Gorlick, M.M., Taylor, R.N., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D., and Wolf, A. *An Architecture-Based Approach to Self-Adaptive Software.* IEEE Intelligent Systems 14(3):54-62, May/Jun. 1999.

[6] Shang-Wen Cheng et al, *Software Architecture-based Adaptation for Pervasive System.* International Conference on Architecture of Computing Systems (ARCS'02): Trends in Network and Pervasive Computing, April 8-11, 2002.

[7] Manuel Roman, Christopher K. Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. *Gaia: A Middleware Infrastructure to Enable Active Spaces,* In IEEE Pervasive Computing, pp. 74-83, Oct-Dec 2002.

[8] Robert Grimm, et al. *Programming for pervasive computing environments.* Technical report UW-CSE-01-06-01, University of Washington.

[9] Peyman Oreizy and Richard N. Taylor. *On the Role of Software Architectures in Runtime System Reconfiguration,* Peyman Oreizy and Richard N. Taylor. Proceedings of the International Conference on Configurable Distributed Systems (ICCDS 4). 1998.

[10] Peyman Oreizy, David S. Rosenblum, Richard N. Taylor. *On the Role of Connectors in Modeling and Implementing Software Architectures.* Technical Report UCI-ICS-98-04, Department of Information and Computer Science, University of California, Irvine, February 1998.

[11] Steven D. Gribble et al. *The Ninja Architecture for Robust Internet-Scale Systems and Services,* To appear in *Special Issue of Computer Networks on Pervasive Computing*