Technique-based Programming

Justin Mazzola Paluska, Hubert Pham, Umar Saif,
Man-ping Grace Chau, Chris Terman, and Steve Ward
{jmp,hubert,umar,mpchau,cjt,ward}@mit.edu
MIT Computer Science and Artificial Intelligence Laboratory
Cambridge, MA, U.S.A.

"*It is not the strongest of the species that survive, nor the most intelligent, but the ones most responsive to change.*"

—Charles Darwin

Abstract

The adaptivity demands of ubiquitous computing motivate applications structured around implementation choices that are (a) made at runtime, reflecting available resources; (b) re-evaluated and potentially changed during application operation, accommodating failures and resource discovery; and (c) locally extensible beyond choices anticipated (and fixed) by central application code.

To this end, we explore a new organization for adaptive applications involving the formalization of *Goals* as representations of abstract services and a universe of *Techniques* which compete to provide them. In our model the selection of Techniques to satisfy each Goal is performed by an application-generic runtime process which continues so long as the Goal remains active, allowing applications to adapt to runtime changes in available resources. The set of available Techniques is potentially open, distributed, and incrementally extensible, providing paths for the decentralized evolution of applications.

1 Introduction

Applications in dynamically changing computing environments must be able to continuously adapt to changes in user locations and needs, respond to component failures and newly available resources, and maintain continuity of service despite changes in available resources. Such behavior necessarily requires frequent re-evaluation of available alternatives, as well as heuristic compromises to best address the application needs with imperfect resources.

Consider, as a simple example, the design of a distributed, adaptive music player. At an abstract level, the music system contains a music stream source, a music stream sink, and a user interface that can control both the source and the sink. True run-time adaptivity in this application requires more than simply selecting three compatible components; the application might be expected to alter its connection topology to suit available network connections. Thus, the system must be able to choose between alternative implementation plans, based on properties of devices discovered in the run-time environment.

While ad-hoc application logic that chooses among different top-level application structures can of course be implemented using conventional application models, the complexity (and hence

fragility) of the resulting code grows in proportion to the number of combinations it must anticipate. As computing environments become richer, conventional models quickly become complicated and hard to maintain.

The rest of this paper explores a systematic way of creating adaptive applications that we call Technique-based programming. Our system provides an application-generic framework for making, monitoring, and revising application-specific implementation decisions automatically.

2 Techniques and Goals

Our approach revolves around the identification of certain critical implementation decisions (e.g. the choice of a suitable sound source in a music system), formalizing these decisions using stylized programming constructs, and arranging for them to be made automatically by an application-generic runtime planning process. As latter process runs concurrently with the application itself, changes in the environment can cause these automatic decisions to be revised *during operation* of the running application. Decisions thus automated follow a common pattern: the choice of one of several alternative *techniques* to satisfy some common *goal*. Their formalization involves two corresponding program constructs: Goals and Techniques.

2.1 Goals

A *Goal* is similar to a disembodied generic procedure call: it uses a name to identify a service, and a number of typed parameters to refine selected details. The Goal PlayMusic(genre="jazz") uses the Goal name PlayMusic to identify the general service to be performed and the parameter genre to narrow its choice of source music. The process of *satisfying* a Goal involves taking some set of steps necessary to provide the service associated with that Goal.

Although short names (like PlayMusic) are used to identify Goals here, each name resolves to a URI that serves as a globally unique identifier for that Goal. The URI points to an XML description containing a mix of formal and informal documentation for that Goal, the semantics of its parameters, properties required of any code that claims to satisfy the Goal, and a *satisfaction* expression to evaluate Techniques satisfying the Goal. The satisfaction expression serves as a Goal-specific evaluation metric for selection of Techniques, allowing our application-generic runtime system to make Goal-specific decisions.

2.2 Techniques

Each Technique is a code module that provides one alternative approach to satisfying a particular Goal. The Technique specifies what Goal it might satisfy, subgoals it requires, and code to be executed if it is chosen. In return, Techniques must publish key-value pair *Properties* that describe the quality of the service that they will provide.

Although Techniques declare what Goal they might satisfy, there is no other physical or logical binding between Goals and relevant Techniques; thus the universe of Techniques applicable to a Goal can be expanded without change to (or even anticipation by) that Goal.

Figure 1 shows a sample PlayMusic Technique that routes music through a visualizing GUI. Its header identifies the Technique and the Goal that the Technique satisfies. The header is followed by the Technique body, which comprises a series of *evaluation stages* followed by a series of *exe-*

cution stages. Each evaluation stage contains a small amount of code that explores the current computing environment. Our runtime system uses the evaluation stages to incrementally construct the execution requirements of the Technique and estimate how well the Technique can satisfy its goal. The execution stages simply provide the code that is run when a Technique is chosen by the Planner for execution.

2.3 Planner

The Planner is the Technique-oriented runtime system. It is a middleware layer that sits between users/applications and an existing, lower-level component system.

The Planner maintains the list of active Goals and doggedly tries to find and instantiate the best Techniques to satisfy those Goals. Applications assert Goals using the `satisfy(Goal, parameters)` API call; such a Goal persists until the application revokes it. Each Goal asserted by a top-level satisfy() call becomes the root node of a *Plan Tree*; this Plan Tree encodes all of the possible alternatives for the Goal that the Planner is exploring.

The Planner transforms a Goal into a working implementation using a four-step Goal satisfaction process:

Build The Planner constructs a Plan Tree for each active goal by finding Techniques for each Goal and recursively satisfying the subgoals of each Technique.

Evaluate Next, the Planner makes a heuristic choice of the "best" Technique for each active Goal by evaluating the Properties of the Technique in the context of the Goal's satisfaction expression.

Execute Once there is a viable way to satisfy the Goal, the Planner runs the execution stages for the chosen Techniques. Execution starts from the bottom the Plan Tree and proceeds towards the root of the tree.

Monitor and Update After execution, the Planner keeps running the evaluation stages of its active Techniques to keep Property values updated as the computing environment changes. If, for a particular Goal node, the best Technique changes from A to B, the Planner shuts down the subtree under A and executes B's subtree.

The Goal satisfaction algorithm is generic, but tuned to application-specific needs by the subgoals that Techniques declare, the Properties that Techniques export, and the Goal-specific satisfaction calculation.

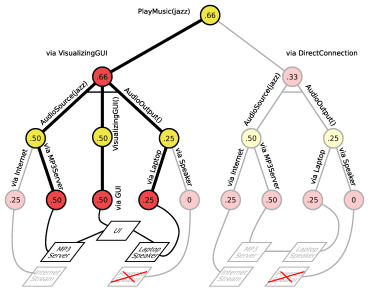
2.4 PlayMusic: A Distributed Application

As an example, consider our distributed audio application from before. In a Technique-based implementation, we represent the user's high-level intent to play music with the PlayMusic goal. When the user desires to hear jazz, she asserts an instance of this Goal with the genre set to jazz, e.g., PlayMusic(genre="jazz").

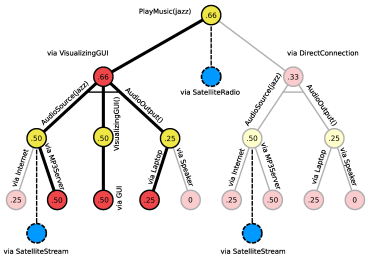
The Planner recursively constructs the Plan Tree for application, resulting in Figure 2. At the root of the tree is the PlayMusic goal instantiated with genre=jazz. The Goal is satisfied by several Techniques, including the via VisualizingGUI Technique of Figure 1. Each of the AudioSource,

```
to PlayMusic(genre)
  via VisualizingGUI:

  subgoals:
    source = AudioSource(goal.genre)
    sink = AudioSink()
    ui = VisualizingGUI()

  eval:
    solution.bitrate = source.bitrate
    solution.satisfaction =
        average(source.satisfaction,
                ui.satisfaction,
                sink.satisfaction)

  exec:
    connect(source, ui)
    connect(ui, sink)
    source.start_playing()
```

Figure 1: A Sample Technique



(a) Snapshot when the via Visualizing-GUI Technique is chosen



(b) Potential ways a satellite radio can be introduced to the PlayMusic Plan Tree.

Figure 2: Example Plan Trees. Goal nodes are in yellow. Techniques are in red and have their subgoals tied by cross-bars. All nodes have their satisfaction values written inside their nodes and nodes chosen for execution are emphasized.

AudioSink, and VisualizingGUI subgoals are recursively satisfied by devices and software found in the environment by the component system.

Initially, the subgoals are satisfied by Techniques that wrap an MP3 server, a local piece of GUI code, and a laptop speaker; each chosen because it has a higher satisfaction than its siblings. Suppose a previously unavailable device becomes available, e.g., the user comes in range of powered speaker. Since the Planner continually monitors its Plan Trees, when the Planner receives a "new device" notification from the component system, the Planner wakes up the dormant via Speaker Technique and re-evaluates it. The Planner updates the tree and, if the device is superior to devices already available, incorporates the new device into the running implementation, either by shutting down the entire tree and restarting, or by hot-swapping if the via VisualizingGUI Technique supports hot-swapping.

Now suppose that a new, unanticipated device, such as a satellite radio tuner, is brought into the user's system. After the component system informs the Planner of the satellite radio's availability, the Planner tries to find Techniques for it. Techniques may come from Internet-based searches, a URI supplied by the device, or even be embedded in the device itself. As long as the Planner can read the Technique, it may make use of it. For example, one Technique may provide a AudioSource interface to the satellite radio, while another may directly provide a PlayMusic interface that is more aware of the tuner's capabilities and programming options (see Figure 2(b). In either case, the Techniques are integrated into the Plan Tree and may be chosen if they are more satisfactory than the existing ones.

3 Architectural Discussion

The Technique-based programming model explored here involves subtle differences from conventional approaches to application programming, with consequent impact on the process of software development and the management of its evolution.

3.1 Goals as commodities

Each Goal serves to standardize the packaging of a service in a way that is understood by both caller and callee, allowing meaningful evaluations and comparisons to be made of alternative options. To promote this commoditization, our unique identifier for each Goal is a URI pointing to an XML specification documenting the semantics of the Goal. Goal specifications include both human-readable *man*-like documentation for Technique programmers as well as machine-readable introspection information used by the Planner. Figure 3 illustrates one particular Goal specification, for a `signal` Goal.

The specifications perform four critical functions: (1) they (by virtue of their URI) provide a readily expandable universe of unique identifiers for Goals; (2) they document the associated service, ideally to a level where *every* Technique that satisfies a given goal will be serviceable in an application demanding that Goal; (3) they define the semantics of the parameters given to the Goal and Properties expected by Techniques; and (4) provide an satisfaction expression that renders Technique Properties into a real number to enable comparison.

Standardization of the satisfaction expression at the Goal level seems reasonable, since satis-

```
<?xml version="1.0" standalone="yes" ?>
<goal shortname="SignalUser">
 <documentation>
  <brief_description>
    A <this /> goal tries to get the attention of a user.
  </brief_description>
  <long_description>
   The importance level may be used by the receiving user to determine
   what kind of notification should be used.
  </long_description>
 </documentation>
 <parameters>
 <parameter>
  <name>user</name>
  <type>http://o2s.csail.mit.edu/specs/rtypes/Person.xml</type>
  <documentation><brief_description>
   The user to be signaled.
  </brief_description></documentation>
 </parameter>
 <parameter>
  <name>max_delay</name>
  <type>http://o2s.csail.mit.edu/specs/rtypes/float.xml</type>
  <default_value>0</default_value>
  <documentation><brief_description>
    The maximum acceptable delay, in seconds, before the user
    is notified.
  </brief_description></documentation>
 </parameter>
 <parameter>
  <name>preferred_delay</name>
  <type>http://o2s.csail.mit.edu/specs/rtypes/float.xml</type>
  <default_value>0</default_value>
  <documentation><brief_description>
    The maximum delay, in seconds, that perfectly satisfies
    this goal.
  </brief_description></documentation>
 </parameter>
 </parameters>
 <properties>
   <property>
     <name> expected_latency </name>
     <type>http://o2s.csail.mit.edu/specs/rtypes/float.xml</type>
     <documentation><brief_description>
     The expected time between request and notification of user, in
     seconds.
     </brief_description></documentation>
 </properties>
 <satisfaction>
   1 if preferred_delay > expected_latency else
     FAILURE if  expected_latency > max_delay else
       (expected_latency - preferred_delay) /
         (max_delay - perferred_delay)
 </satisfaction>
</goal>
```

Figure 3: The XML Goal specification for the `signal` Goal. For readability, we've left special XML characters unescaped in the satisfaction expression.

faction must (a) reflect Goal-specific detail and (b) be comparable across competing Techniques. Our current expression language uses the expression subset of Python 2.5 [1]; the subset includes the standard arithmetic and comparison operators, as well as set membership tests, and an infix conditional [2]. We also provide a `FAILURE` constant that subsumes other values; if a Technique's satisfaction evaluates to `FAILURE`, the Planner stops exploring it.

The signal Goal of Figure 3 is parameterized by a preferred delay and a maximum acceptable delay in delivering a notification to a user. Techniques satisfying the signal Goal need to export their expected notification latency. If a Technique can meet the preferred delay, it is rated with a perfect satisfaction of 1. If a Technique cannot make the maximum allowed delay, then the satisfaction expression evaluates to `FAILURE` and the Planner ceases to explore the Technique. Finally, for Techniques with latency between the preferred delay and the maximum delay, the satisfaction function is a linear scale.

3.1.1 Truth in (Property) Advertising

In our current implementations there is no mechanism for enforcement of the behavior promised by a Technique. A selected Technique may fail to deliver its promised level of satisfaction; in the extreme, it may intentionally misrepresent the function it performs (much as conventional library procedures might). Like library procedures, Techniques must be trusted code.

Even if we rule out malicious Techniques, however, reported Properties (and the derived satisfaction) are merely estimates of expected performance. Different Techniques addressing a Goal may differ in the accuracy of their predictions, potentially biasing the planner toward those Techniques that consistently overestimate their Properties. A buggy Technique that consistently promises highly-satisfactory Properties and fails to deliver usable service can cause a fatal failure in our current system, despite mechanism for failure detection and re-evaluation of the Plan Tree. Each subsequent evaluation will select the same buggy Technique, detect its failure, re-evaluate the plan tree, and repeat.

There are a variety of plausible ways to improve on this situation. The actual satisfaction delivered by a Technique can be measured, since it is algorithmically determined from measurable parameters. Thus, the disparity between the satisfaction delivered and that promised by a Technique can be measured, enabling mechanisms for biasing future decisions against Techniques that tend to predict satisfaction optimistically. Even a crude learning mechanism of this kind could break the persistent failure loop cited above, as repeated failures of a buggy Technique would eventually bias the Planner to replace it with some (presumably better) alternative.

3.1.2 Immutability

Once a goal has been published, it serves as a public interface point. Its URI may be built into Techniques offering to provide the represented service, and requiring (as subgoals) the service to be supplied. Of course, semantic changes in the goal will, in general, render these dependencies obsolete; consequently, we require that Goals (and hence their specifications) be immutable.

In general, the evolution of a Goal's specification requires that a new specification (having a distinct URI) be created. The new specification may note it as a replacement for the old Goal, that the latter is now deprecated, or even that the new version is strictly narrower than the old (in the

sense that any Technique satisfying the new Goal is guaranteed to satisfy its predecessor). Existing Techniques citing the old Goal will continue to use the (possibly deprecated) version until updated, although it is plausible that such updates could be automated in certain cases.

3.2 Techniques and code refactoring

Technique-based programming may be viewed as an alternative to conventional procedural or object-oriented approaches to system organization. Our deliberate separation of each Technique addressing some given Goal into an independent code module, however, defies conventional procedural and object-oriented programming practice which tends to localize code performing closely related functions.

Consider, for example, a conventional application addressing the PlayMusic goal of Section 2.4. Some module within that conventional application, say a FindMusic(...) procedure or method, might search various plausible sources (cached MP3s, Internet sites) for available tunes, applying some heuristic model of user tastes to select the program material to play. While FindMusic may be arbitrarily complex and sophisticated, investigating a variety of known music sources, it represents a single repository for all algorithms to be use by the application to find music. Expansion of the function it performs requires changing the code in that module.

Suppose there are several alternative approaches taken by FindMusic under various circumstances. We might imagine the module coded with a top-level multi-way conditional branch, e.g. Figure 4, whose logic delegates the task at hand to code segments specific to each approach. The latter segments may themselves be encapsulated as individual modules; the FindMusic module then serves primarily to (1) identify the applicable approaches and (2) provide logic for selecting among them. The approach-specific modules may themselves encapsulate multiple sub-approaches; for example, find_net_music may have ways of organizing baroque music that are distinct from those used for heavy metal. Thus they may themselves have top-level conditional logic selecting between code appropriate to each sub-approach, which may itself be similarly hierarchical. In conventional systems, this hierarchy of conditionals may be explicit, like that of Figure 4, or hidden behind object factories, virtual function calls, or generic procedures.

The decision tree that reflects the resulting hierarchy of conditionals is essentially a Plan Tree: modules containing N-way branches among available alternatives correspond to Goals, while the modules implementing each approach are roughly the equivalent of Techniques. From this perspective, the distinction between a Technique-based approach and stylized conventional procedural code may seem a superficial refactoring. The important difference is however, that the conventional tree requires code modules that enumerate, and hence constrain, the set of alternatives at each decision point. Our approach avoids any such explicit enumeration: the set of available Techniques is open, potentially decentralized, and can be locally extended simply by writing new Techniques.

This difference induces two unique architectural characteristics of our system. They include its principal advantage and major complication relative to conventional program organization: (1) the local extensibility of the Technique set, and (2) the need for implicit decision logic. Each of these issues is discussed in a following section.

3.2.1 Technique extensibility

Our deliberate avoidance of constraints on the set of Techniques applicable to each Goal supports a conceptual model of that set as a strictly "additive" universe. Each Technique describes a way of achieving some Goal, given the appropriate prerequisites. In effect it proposes a contractual relationship: if the caller supplies the prerequisites and takes the specified actions, the Technique promises to satisfy the advertised Goal. Abstractly, then, a Technique never becomes obsolete. It encodes a way of doing something — a way which may become unused (either because its prerequisites become unsatisfiable or because competing Techniques promise better results) but is never "wrong."

An advantage of this additive universe is that we can extend its behavior without changing any existing code, but by simply by making new Techniques available.

3.2.2 Implicit decision logic

The principle disadvantage of our additive universe of Techniques is that there is no explicit logic dictating the choice between competing approaches. Indeed, in our model even a venue for such code is deliberately avoided: nowhere in our code is there a construct that encapsulates every way of satisfying a given Goal, since such a construct would interfere with the extensibility advantages discussed in the previous section.

For meaningful choices among competing Techniques to be made by an application-generic planner, the satisfaction values computed for each Technique need to reflect some common metric. One major architectural implication of our approach is that instead of localized, Technique-specific (and hence Goal-specific) mechanism for deciding among competing approaches to satisfying a Goal, we must use a generic (Goal- and Technique-independent) decision algorithm relying on information input from each of the independently-written competing Techniques.

3.3 Can't get no Satisfaction? Find Suitable Techniques.

Suppose our PlayMusic example environment includes music sources and destinations having format requirements that must be matched. A primitive attempt to accommodate this requirement is sketched in Figure 5, which shows a Technique whose subgoals include an audio source and an audio sink.

This Technique compares the format attributes of its proposed source and sink solutions, and signals the unacceptability of a mismatch via an explicit call to fail. This call is tantamount to setting the Satisfaction property to zero: without arguments, fail() terminates exploration of the corresponding subtree.

Often this declaration of failure is too radical. Our search for a satisfactory solution to this particular goal will provide exactly one set of bindings for the specified subgoals; if their formats are incompatible, the use of this Technique — with *any* other possible bindings of the subgoals — is abandoned. In the present example, there may be source/sink pairs with compatible formats which will be neglected simply because the pair reflecting the highest Satisfactions happened to be incompatible. Thus, the approach represented in Figure 5 suffers from a combination of deficiencies: (1) that Satisfaction values computed for subgoals (and hence the choice of ways to satisfy each

subgoal) do not reflect the critical format-matching requirement; and (2) that a single bad choice of subgoal approaches will occlude the exploration of lower-rated but potentially viable solutions.

3.3.1 Backtracking

From a performance standpoint, the second of these restrictions — the go/no-go evaluation of a Technique node based on computed satisfaction — is laudable: final decisions are made in a single bottom-up pass, visiting each Plan Tree node at most once. Given the Technique coding of Figure 5, however, this efficiency comes at the cost of missing potentially viable solutions. We may moderate this compromise by passing optional arguments to fail, giving hints to the planning process that encourage it to explore re-evaluation of the current Technique node with alternative bindings of its subgoals. At the opposite extreme of our thoroughness/efficiency tradeoff, we may specify `fail(explore_alternative=True)` which causes *every* combination of plausible (non-failing) subgoal solutions to be explored, in order of decreasing Satisfaction, until some binding causes the current node to succeed with an acceptable satisfaction.

Unconstrained use of this option leads to full *backtracking* and the exponential worst-case performance costs that doomed Prolog-era AI search strategies. If all alternatives are explored at every level of the tree, the search of the solution space effectively enumerates and evaluates every plausible combination of Techniques until some acceptable configuration is found. While a limited amount of searching is tolerable, a more selective approach to the search is a practical necessity for all but toy examples.

3.3.2 Search-narrowing Goal parameters

One simple alternative to the exhaustive search involves making decisions high in the search tree and passing search-narrowing parameters down the tree for each subgoal. Figure 6 sketches a revised search for a source and sink, each specifying wav as the media format (restricting solutions to devices that accept or emit this media type). If multiple formats are to be explored, this approach requires that an alternative node be established for each plausible format. This can be done via explicit format-specific Techniques (as above), or by using mechanism introduced in the next section to chose a format in an early phase of the Technique and pass it as subgoal parameters in a subsequent one.

3.3.3 Sequential subgoal binding

Rather than performing parallel searches for all ways of satisfying each subgoal followed by a check for the consistency of the solutions, we may improve search performance by ordering the subgoal searches. For example, we might (1) search for a source accepting arbitrary format, and then (2) search for a sink whose format is compatible with that of the source we've found. To that end, we allow multiple subgoals: stages within a single Technique. They are executed in sequence, and attributes of subgoal bindings from earlier segments may be used to direct searches in subsequent ones. Figure 7 illustrates the use of this mechanism to constrain the search of our example.

```
if internet_connected() then
   return find_net_music("http://music.org", ...)
else if exists("/usr/music_lib") then
   return find_music_from_lib("/usr/music_lib")
else ...
```

Figure 4: Example of traditional component selection code

```
to PlayMusic(genre):

  subgoals:
    source = AudioSource(goal.genre)
    sink = AudioSink()

  eval:
    if source.format != sink.format:
       fail()
    else: solution.satisfaction = ...

  exec:
    ...
```

Figure 5: Explicit failure

```
to PlayMusic(genre):

  subgoals:
    source = AudioSource(goal.genre,
                         format="wav")
    sink = AudioSink(format="wav")
```

Figure 6: Constraints passed down the tree

```
to PlayMusic(genre):

  subgoals:
    source = AudioSource(goal.genre)

  subgoals:
    sink = AudioSink(format=source.format)
```

Figure 7: Ordered subgoal bindings

4 Implementation

We implemented the Planner in pure Python and have tested it on GNU/Linux, Apple OS X, and Microsoft Windows (under cygwin). Full source code to the Planner is available under a free license from `http://o2s.csail.mit.edu/`. A small library of Goal specifications (as well as an HTML interface to the specs) can be found at `http://o2s.csail.mit.edu/system.html`.

4.1 The Planner

We took a lesson from Python and made our implementation both easy to run from the command line and embed in other applications in order to encourage use of the Planner in legacy applications.

Internally, the Planner module is similar in design to single-threaded GUI and network toolkits like GTK [3] and Twisted [4]. In those toolkits, the application is built around a single event loop that surveys the work that needs to be done and schedules the work for execution. In the Planner, each Technique with evaluation stages that can be run represents a unit of work and a scheduler determines which Technique will be the next to run. The scheduling policy the Planner modules uses is pluggable. Our simplest scheduler, the QueueScheduler, schedules nodes as they need updates without regard to their position in the Plan Tree. The remaining three explore three different ways of delaying updates to nodes higher in the tree until nodes further down in the tree have stabilized.

Our Planner implementation provides a powerful primitive for Plan Tree manipulation, Technique cloning. When a Technique is cloned, the Planner makes a copy of Technique in its Plan Tree for every subgoal binding. Cloning enables the planner to consider all combinations of subgoal bindings. For example, Figure 8 illustrates node cloning of the via DirectConnection Technique in our PlayMusic example. The via DirectConnection Technique has two subgoals (AudioSource and AudioSink), each satisfied by two Techniques, leading to four combinations for consideration. By cloning and exploring all combinations of sub-goal bindings rather than simply binding Techniques to sub-goals independently, the Planner can potentially exploit situations where specific combinations of sub-goal bindings can yield higher satisfaction than the sum of the parts bound independently.

We use Technique cloning in two places. First, we invoke cloning when a Technique calls fail() and requests that the Planner explore alternative subgoals. Second, we provide a GenerateSolution primitive Goal that allows a single Technique to be associated with more than one Solution object. The Planner creates a clone in its Plan Tree for each Solution associated with the Technique.

4.2 Technique Evaluation Stages

Programmers write each stage of the Technique as though all other stages have completed, and when the last evaluation stage completes, the Planner considers the Technique ready for execution. The Planner supports this model by maintaining a current_stage pointer for each Technique; the pointer is incremented whenever a stage successfully completes. This model allows programmers to incrementally refine Properties of the Technique in program text order and signal to the Planner when all relevant evaluation is complete. It also allows the Planner to partially evaluate some Techniques, abandon exploration, and restart exploration later.

However, this model is complicated by the fact that Properties are not static in nature: If

Technique stages are evaluated in text order without regard to external influences, Property values calculated later in the Technique may rely on stale data. Thus, to allow Technique to refresh stale data while maintaining the sequential programming nature of Techniques, the Planner can "roll-back" the Technique to an earlier evaluation stage and re-execute all subsequent stages.

In order to provide rollback seamlessly, the Planner keeps track of the Property dependencies and pre-execution state for each stage in each Technique. We also require that Technique evaluation stages be idempotent. When a Property p (e.g., a subgoal Property) that a Technique depends on changes, the Planner finds the first stage s that references that Property, resets `current_stage` to s, and resets the Technique's internal state to what it was the last time s was just about to be executed. Thus reset, the Technique is re-evaluated by the Planner.

5 Evaluation

5.1 "JustPlay" Demo Implementation

We implemented all of the components and Techniques required to implement the `PlayMusic` goal [5] as a real-world test of our system. The Planner runs on top of the O_2S NPOP component system [6]. Users interact with the system by speaking commands into a Nokia 6680 cell phone. The cell phone is in turn connected to a voice recognition system [7] that converts user commands into Goals. In turn, these Goals are fed into our Planner.

We implemented Techniques for five Goals, roughly corresponding to the Plan Tree of Figure 2. These Techniques vary from "algorithmic" Techniques that simply outline a connection topology to Techniques tuned to a particular piece of hardware, like the Nokia phone or several speakers specially modified with 802.11 network interfaces.

For comparison, we also implemented the same application logic using ad-hoc code. The Technique-based approach and the ad-hoc approach both require about the same number of lines of code. However, the Technique-based approach is much more extensible — we can add a new implementation strategy by simply adding new Techniques.

5.2 Performance

The Planner does not interpose itself in the data streams between individual components, so it does not slow down an application once it is running. However, the Planner necessarily takes part in application start-up and adaptivity since the Planner drives the decision making of these phases. We evaluate our system in order to determine how much latency does the Planner introduces into a running application.

All tests are run on a desktop Pentium 4/3.2GHz with 1 GB of RAM running GNU/Linux 2.6.8.

Experimental Setup We generated a set of stub Techniques that induce large Plan Trees of 4 sizes. Each Goal in our test setup is satisfied by two Techniques; Each Technique—save the Techniques at the bottom layer—declares two subgoals. We vary the depth D of the trees from 1 "Goal-Technique" layer to 4. The Techniques contain no evaluation or execution code, so the measured execution time of the Planner is solely due to Planner overhead.

Our start-up latency test measures how long it takes the Planner to build, evaluate, and execute trees of various depths. After the Planner has converged on a solution and idled, we introduce a

variety of satisfaction changes to the leaf nodes to simulate failures. Our swap latency test measures how long the Planner takes to converge to a new plan after the failures are introduced. We run the experiment for each of our three schedulers.

Results Figure 9 summarizes our latency tests. Overall, for trees with tens of nodes the Planner is able to start-up in a few seconds and adapt in less than a second. While our start-up time seems high, in practice we find that latency of interacting with remote devices over the network dwarfs the latency of the Planner. On trees with hundreds of nodes, the Planner is slow; however, we have yet to encounter such large trees in real world applications.

Figure 9(c) shows how long the Planner takes to react to changes in its environment and swap in new Techniques. Note that this does not measure average application downtime, but rather how long the Planner takes to determine what changes need to be made and then instantiate those changes. Typically, application downtime is lower than total swap time because the Planner runs in the background until the final switch over to the new Techniques.

As Figure 9(a) shows, the Planner's start-up latency can be reduced with a smart scheduler that reduces the amount of work the Planner must do. The QueueScheduler has the worst performance of the three schedulers by a factor of 2. This is primarily because the QueueScheduler takes approximately twice as many iterations to converge to a stable solution than the other schedulers. The result makes sense because each node gets scheduled every single time a child node updates itself, leading to repeated work. The PrecedentQueueScheduler, FavorBest, and TreeScheduler behave similarly because they both avoid repeating work at higher levels in the tree until lower levels have stabilized. The PrecedentQueueScheduler has a slight speed advantage since its time complexity is linear in queue size and the queue is usually much smaller than the entire tree. Future schedulers may completely avoid working on certain plan tree branches, further reducing work and increasing performance.

On the other hand, as 9(c) shows we find that fail-over latency is less dependent on the choice of scheduler than it is on the overall size of the Plan Tree. This is mainly because there are fewer nodes to schedule and also because most of the fail-over time is spent shutting down the old Techniques and starting the new ones, both of which do not involve a node scheduling strategy.

6 Related Work

Many systems provide abstractions that ease the burden of programming adaptive applications. At the network level are systems like MIT's Intentional Naming System [8], Service-oriented Network Sockets [9], and Lightweight Adaptive Network Sockets [10]. These systems allow applications to opportunistically connect to the best resources in a given environment. Similarly, at the application level, CMU's Aura system [11] and UIUC's Gaia [12] provide high-level programming abstractions that encoding common tasks in pervasive computing environments. PCOM [13] furthers this idea by modelling components with contracts the runtime system to explore component requirements. Our system builds on these systems by providing competition between approaches to solving a problem as well as competition between resources in the environment.

Semi-automatic service composition systems such as NinjaPaths [14] or SWORD [15] complement our approach. Such systems can be used to generate Techniques and aid in rapid development of

Technique-based applications.

The level of programming abstraction advocated by our system also shares its motivation with declarative and implicative programming approaches, especially rule-based systems. A Goal, much like a rule in a rule-based system, such as Clips [16] or Jess [17], encodes a pattern to be matched against a set of solutions. However, instead of requiring a programmer to express the entire application logic as, for instance, a set of first-order predicate logic rules, our approach permits embedding of imperative code inside a Technique. This is similar to systems like InterPlay [18], which uses a hybrid rules/code engine based on Jess to compose tasks within a consumer electronics environment.

7 Conclusion

Emerging computing environments require new abstractions that permit increased levels of runtime adaptivity. To this end, we propose a set of abstractions whereby applications make implementation choices during runtime and hence take into account resource availability. The system continuously re-evaluates and potentially changes these implementation decisions, enabling adaptivity in both the application's resources and structure during runtime. Because the set of implementation choices is distributed and yet locally extensible, the system enables the decentralized evolution of applications.

To realize these abstractions, we formalize implementation decisions using stylized programming constructs and arrange for them to be made automatically by an application-generic runtime. As a result, we develop the Technique-based, Goal-oriented programming framework: Goals name required services and Techniques implement those services. The framework fulfills the above abstractions by providing (1) mechanism to let independently developed code modules and resources compete to satisfy high-level system requirements and (2) data structures — Plan Trees – that embody the implementation choices made by the Planner and permit revision of the choices as runtime conditions change.

References

[1] Python Software Foundation. Python 2.5. `http://www.python.org`.

[2] Guido van Rossum and Raymond D. Hettinger. Conditional expressions. Technical Report PEP 308, Python Software Foundation, 2006.

[3] GTK+: The GIMP Toolkit. `http://www.gtk.org/`.

[4] Twisted Matrix Laboratories. Twisted Python. `http://twistedmatrix.com/`.

[5] Justin Mazzola Paluska, Hubert Pham, Umar Saif, Chris Terman, and Steve Ward. Reducing configuration overhead with goal-oriented programming. In *PerCom 2006: Works in Progress*, March 2006.

[6] Hubert Pham. A distributed object framework for pervasive computing applications. Master's thesis, Massachusetts Institute of Technology, 2005.

[7] S. Seneff, E. Hurley, R. Lau, C. Pao, P. Schmid, , and V. Zue. Galaxy-ii: A reference architecture for conversational system development. In *Proc. ICSLP '98*, November 1998.

[8] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *SOSP99*, pages 186–201, 1999.

[9] Umar Saif and Justin Mazzola Paluska. Service-oriented network sockets. In *MobiSys 2003*, 2003.

[10] Umar Saif, Justin Mazzola Paluska, and Vijay Praful Chauhan. Practical experience with adaptive service access. *SIGMOBILE Mob. Comput. Commun. Rev.*, 9(1):27–40, 2005.

[11] David Garlan, Daniel P. Siewiorek, Asim Smailagic, and Peter Steenkiste. Project aura: Toward distraction-free pervasive computing. *IEEE Pervasive Computing*, pages 22–31, April-June 2002.

[12] Anand Ranganathan, Shiva Chetan, Jalal Al-Muhtadi, Roy H. Campbell, and M. Dennis Mickunas. Olympus: A high-level programming model for pervasive computing environments. In *PerCom 2005*, March 2005.

[13] Christian Becker, Marcus Handte, Grego Schiele, and Jurt Rothermel. Pcom — a component system for pervasive computing. In *PerCom 2004*, pages 67–76, March 2004.

[14] Sirish Chandrasekaran, Samuel Madden, , and Mihut Ionescu. Ninja paths: An architecture for composing services over wide area networks. http://ninja.cs.berkeley.edu/dist/papers/path.ps.gz, 2000.

[15] Shankar R. Ponnekanti and Armando Fox. Sword: A developer toolkit for building composite web services. In *The Eleventh World Wide Web Conference (Web Engineering Track)*, 2002.

[16] NASA. Clips reference manual. In *NASA Technology Branch, 1993*, October 1993.

[17] Sandia National Laboratories. Jess rule-based system. In *Jess User Manual*, 2003.

[18] Alan Messer, Anugeetha Kunjithapatham, Mithun Sheshagiri, Henry Song, Praveen Kumar, Phuong Nguyen, and Kyoung Hoon Yi. Interplay: A middleware for seamless device integration and task orchestration in a networked home. In *PERCOM'06*, pages 296–307, March 2006.

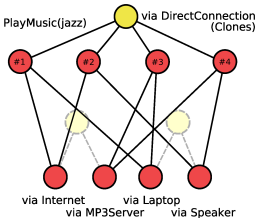
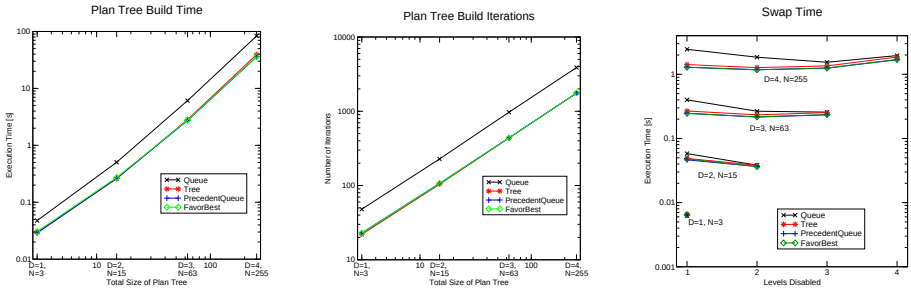


Figure 8: Plan Tree with cloned Techniques



(a) Start-up Time (b) Start-up Iterations (c) Swap Times

Figure 9: Planner-induced latencies for four different tree sizes. D indicates the depth of the tree; N, the total number of nodes in the tree is included for reference. In 9(c), we cluster results by the size of the tree.