Structured Decomposition of Adaptive Applications

Justin Mazzola Paluska, Hubert Pham, Umar Saif,*
Grace Chau, Chris Terman, and Steve Ward
{jmp,hubert,mpchau,cjt,ward}@mit.edu
MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, U.S.A.
*umar@lums.edu.pk
LUMS Computer Science Department, Lahore, Pakistan

Abstract

We describe an approach to automate certain high-level implementation decisions in a pervasive application, allowing them to be postponed until run time. Our system enables a model in which an application programmer can specify the behavior of an adaptive application as a set of open-ended decision points. We formalize decision points as Goals*, each of which may be satisfied by a set of scripts called* Techniques*. The set of Techniques vying to satisfy any Goal is additive and may be extended at runtime without needing to modify or remove any existing Techniques. Our system provides a framework in which Techniques may compete and interoperate at runtime in order to maintain an adaptive application. Technique development may be distributed and incremental, providing a path for the decentralized evolution of applications. Benchmarks show that our system imposes reasonable overhead during application startup and adaptation.*

1. Introduction

Ubiquitous and pervasive computing environments are characterized by a richness and heterogeneity of resources far greater than traditional computing environments. In addition to the variety of devices, there is a high turn-over rate as existing devices leave the environment, either due to failure or voluntary withdrawal when new, potentially better, devices enter the environment. Applications executing in ubiquitous environments are expected to discover relevant resources, evaluate resources, and monitor utilized resources for failure.

Therefore, the ability to somehow adapt to new situations is a key requirement of applications in these environments. In particular, such adaptive systems share two main requirements:

1) They must be able to make implementation decisions at runtime, rather than at design-time or compile-time.
2) They must be able to consider new information at runtime and potentially revise previously made implementation decisions.

In traditional applications, these requirements manifest themselves as a monitor loop that discovers changes in the computing environment (such as new components or changes in the status of already known components) and a set of application-specific decision functions that choose what combinations of components are appropriate. While the application programmer may use design patterns, recursive decomposition, or other design techniques to encode decision logic, the problem remains that whatever code ships with the application enumerates all known ways of adapting the program.

1.1. Examples of Adaptive Systems

Much systems-level work in the pervasive and ubiquitous computing field strives to replace application-level decision logic with application-level dependency declarations. In these systems, application programmers declare "what" they need and a runtime system determines "how" to satisfy each requirement. For example, INS [1] and other systems like it [2], [3] provide *intentional* sockets whose descriptions are resolved by the network layer. Intentional sockets provide extensible decision logic because new services can be added that match existing intentional names.

Above the network layer, component systems like PCOM [4] or RUNES [5] allow programmers to declare dependencies as COM- and CORBA-like interfaces while runtime systems discover and match appropriate components to the required interfaces. In PCOM, components can depend on other components — allowing a form of hierarchical decomposition of application

functionality — and provide component-specific code that aids in resource selection.

1.2. Additional Requirements

The approaches to adaptivity represented by these example systems share the characteristic that the range of possible choices — e.g. of candidate devices and ways in which they are used — is embedded in code at either the application or system level. Extension of application behavior requires modifying existing code, which in turn demands privileged access to the code to be modified. The constraint of adaptive behavior to that anticipated by a centrally-maintained codebase limits the evolution of adaptivity, and hence ultimately limits adaptivity itself.

This deficiency led us to realize an additional, subtle, requirement for pervasive applications:

3) **Decision-making logic must be *open-ended*.** Adding a new device or implementation choice to the environment should not require modification of already-existing code in the system.

In the rest of this paper, we explore a model for open-ended decision logic as a means of programming adaptive applications.

1.3. Open-ended Decision Making

Our work focuses on providing application programmers with a way of managing implementation decisions and component writers with an extensible way of expressing particular implementation plans in a way that allows extensible, but domain-specific, evaluation of alternatives.

Our system relies on two main concepts. First, *Goals* explicitly identify certain critical implementation decision points, as well as describe the problem to be solved by the selected implementation choice. Second, *Techniques* are scripts describing alternative ways of satisfying Goals. Techniques serve two purposes: (1) they provide indirection between the known Goals interface and wide variety of hardware and software interfaces we would like to use and (2) they implement domain-specific evaluation code that lets our system compare alternative Techniques.

Our approach offers three salient features:

1) **Hierarchical Decomposition with Extensible Constrained Evaluation** Techniques may declare multiple prerequisite sub-Goals but provide code that constrains how the Planner chooses to satisfy the sub-Goals.
2) **An Additive Universe of Code Modules** New Techniques can be added to the system without needing to change existing code, aiding the introduction of new device classes and new implementation strategies.
3) **Separation of Decision Logic from Components** Techniques are separate from the components they describe, allowing both to evolve independently.

In addition to these points, our architecture includes two details aimed at lowering user-perceived latency: we allow incremental evaluation of decision logic, which permits our system to make decisions on early estimates of component performance, and we cache decision-making, which lets our system react to typical component failures and re-plan in less than 250 ms.

2. Programming Model

Goals are bound to Techniques at runtime by the *Planner*. Application programmers use the Planner to manage adaptive state, while component programmers write Techniques interpreted by the Planner.

2.1. Goals and Goal Properties

A Goal is an abstraction of a parameterized decision point that describes what functionality is needed without specifying how to implement that functionality. The Goal's parameters serve to restrict the semantics of the Goal, e.g., reducing a generic "play any movie" Goal to the playing of a *particular* movie specified by the name parameter. An application asserts a Goal (with bound parameters) when the application needs to have a certain condition maintained by the Planner.

Concretely, a Goal refers to a specification file that describes the formal parameters of the Goal as well as what *Properties* any Technique that satisfies the Goal must provide. Properties are simple key-value pairs that describe qualities of the implementations behind the Goal. The Planner uses Properties to compare Techniques competing to satisfy the same Goal.

We adopt standard procedural syntax for the parameterization and assertion of Goals; thus, a Goal may be viewed as a disembodied generic procedure whose parameters, Properties, and behavior are described by its specification. The assertion of a Goal may similarly be viewed as an invocation of the disembodied procedure, leaving to the Planner the task of locating an appropriate body of code (Technique) to be executed in order to satisfy the Goal.

2.2. Techniques

A Technique is a small script mixing declarative and arbitrary imperative code broken up into a series

```
to PlayMovie(name, language): via RTPStreams:

    ##### Exploratory Stages #####
    subgoals:
        source = RTPAVSource(goal.name, goal.language)
        sink = RTPAVSink()

    eval:
        # check for compatibility
        if (subgoals.source.stream_format not in
            subgoals.sink.supported_stream_formats):
            planner.fail()
    eval:
        # set properties this combination will provide
        props.resolution = min(subgoals.source.resolution,
                               subgoals.sink.resolution)
        props.screen_size = subgoals.sink.screen_size
        props.stream_format = subgoals.source.stream_format
        props.bitrate = subgoals.source.bitrate

    ##### Commit Stages #####
    exec:
        subgoals.sink.resource.enqueue(uri=subgoals.source.uri)

    update source from old_source:
        subgoals.sink.resource.stop(subgoals.old_source.uri)
        subgoals.sink.resource.enqueue(subgoals.source.uri)

    shutdown:
        subgoals.sink.resource.quit()
```

Listing 1. A Technique that satisfies the PlayMovie Goal by linking an RTP source stream to an RTP output device.

of stages. Techniques are not appropriate for directly implementing application functionality; instead, they are used to wrap existing code modules and resources so that the Planner can use and compare these resources. Listing 1 shows one Technique that satisfies the PlayMovie Goal by connecting an RTP source to an RTP output device.

A Technique's stages may include:

1) **Sub-Goal Declarations** Sub-Goals are sub-decision-points that must be satisfied for the Technique to succeed. They are declared in subgoals stages and provide a simple way of hierarchically decomposing application functionality.
2) **Evaluation Code** eval stages compute the value of the resources or strategies that the Technique represents and export its computations as Properties. eval stages may contain arbitrary code but, since they might be re-run as the environment changes, they must be idempotent.
3) **Commit Code** Commit stages configure and instantiate, update, and shutdown application components.

The stages are run (and potentially re-run) in a schedule determined by the Planner, subject to the constraint that a stage cannot run before all of its predecessors have run at least once. When the last evaluation stage

```
plan = Planner.plan("PlayMovie", name="SimpsonsMovie")
# "plan" is the object containing the Goal Tree for the
# top-level Goal explored in this thread.
while plan.is_running():

    # explore() blocks until a viable Plan is found
    new_snapshot = plan.explore()

    # if the new plan is better or if the current plan has
    # failed, commit to the new plan.
    if is_better(new_snapshot):
        if plan.is_running():
            # update something already running
            plan.update()
        else:
            # Start a new implementation
            plan.commit()

    # Continue to the next iteration of the while loop to
    # see if anything has changed.
```

Listing 2. Application code from a movie player that uses the Planner.

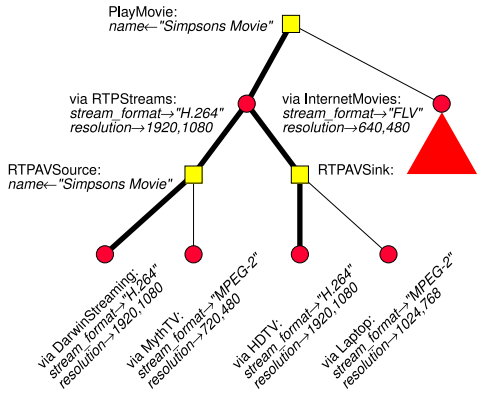

Figure 1. A partially explored Goal Tree for the PlayMovie Goal. Yellow boxes are Goals and show the Goal parameters. Red circles are Techniques and are displayed with their exported Properties. Thick lines represent the chosen Plan for the PlayMovie Goal.

completes, the Planner considers the Technique ready for commitment. Techniques may have many subgoals and eval stages, allowing the Technique programmer to incrementally estimate and refine Property values.

2.3. Goal Lifecycles and the Planner

Applications invoke the Planner and are responsible for deciding when the Planner may make changes to the application's runtime configuration. Listing 2 shows typical application code while Figure 1 illustrates how the Planner expands the PlayMovie Goal.

In Listing 2, an application first asks the Planner to assert a Goal; in return the application gets a handle to the planning process associated with that Goal.

Next, the application asks the Planner to explore() the possible ways of satisfying the Goal.

2.3.1. Goal Exploration. Goal exploration consists of building a *Goal Tree* and evaluating Techniques. The Planner builds the Goal Tree by finding Techniques that satisfy the asserted Goals and recursively matching the sub-Goals of each Technique it finds. Figure 1 illustrates the Goal Tree for a top-level PlayMovie Goal.

The Goal Tree represents all known strategies for implementing the top-level Goal. A "path" from the root Goal node to leaf Techniques represents a particular strategy that implements the Goal. The Goal Tree is an *and-or* tree: a Goal can be satisfied by any child Technique, but a Technique requires satisfaction of all of its sub-Goals. The Planner runs Technique eval stages to extract Properties from each Technique. These properties allow the Planner to heuristically choose a "best" Technique, called the *chosen* Technique, for each Goal. In Figure 1, the bold-faced path shows the chain of chosen Techniques that best implement the top-level PlayMovie Goal. We call this best path the *Plan* for the Goal.

Although the Planner is required to make heuristic choices among the Techniques competing to satisfy each Goal in the tree, it does so by a simple, application-generic process that maps Property values reported by each Technique to a single scalar value; the chosen Technique is simply the Technique that maximizes this value. (Section 3.1 elaborates on this process.) The actual heuristics and application-specific policies are dictated by Goal specifications and Technique code, allowing the evolution of these relatively transient aspects of the system without changes to the Planner or the system architecture surrounding it.

2.3.2. Goal Commitment. If a Plan is found, the application may ask the Planner to commit the Plan. Commitment of the Plan proceeds by running the exec stages of chosen Techniques in a bottom-up fashion. This way, the exec stages of higher-level Techniques can rely on already-configured components supplied by lower-level Techniques. Techniques whose exec stages have been run are said to be *committed*. After commitment, the application may continue to call explore() to cause the Planner to explore, expand, and update its Goal Tree *without* affecting the committed Plan.

2.3.3. Goal Monitoring and Shutdown. Once generated, the Goal Tree serves as a cache of available implementation strategies: startup, failover, upgrade, or shutdown of components in the system simply becomes the activation or deactivation of branches of the Goal Tree. The Planner updates the Goal Tree cache for as long as the top-level Goal is active, permitting rapid re-evaluation of alternative implementations of a Goal throughout the lifetime of the top-level Goal.

The application may also modify the arguments to the Goal to better reflect its changing needs. If a new set of Techniques better satisfies the Goal than the current Plan or if a Technique in the current Plan fails, the Planner notifies the application, which may ask the Planner to upgrade the currently committed Plan. The application has complete control over the upgrade process so that upgrades do not happen at sensitive times.

When the application decides to quit, it tells the Planner to shutdown the Goal: the shutdown stages of committed Techniques are called, and the Goal Tree is garbage collected.

3. Architecture

Our system allows (1) an additive universe of Techniques, (2) appropriate selection of Techniques with inter-dependent sub-Goals, (3) runtime adaptivity, and (4) separation of decision logic from components without restricting the open-ended nature of Goals.

3.1. Additivity

We deliberately avoid constraints on the set of Techniques applicable to each Goal in order to support a conceptual model of that set as a strictly "additive" universe. Each Technique describes a way of achieving some Goal — a way which may become unused (either because its sub-Goals fail or because competing Techniques promise better results) but is never "wrong". An advantage of this additive universe is that we can extend the behavior of our system without changing any existing code, but by simply making new Techniques available.

In order to create our additive universe, our decision-making algorithm must be generic, i.e., it cannot explicitly enumerate and choose Techniques. Instead, our system computes a score called *Satisfaction* for each Technique based on the Technique's self-reported Properties. Thus, Properties can be viewed as the multi-dimensional cost using the Technique and the Satisfaction calculation as a dimension-reducing function to produce a scalar score to allow easy, open-ended competition [6] among alternative Techniques addressing each Goal. The Planner need only choose the Technique with the highest Satisfaction score at each Goal decision point.

Each Goal specification provides a default formula for computing its Satisfaction from Properties reported by Techniques. However, local policy may dictate a different Satisfaction scheme — e.g., users may prefer higher resolution video at a lower frame rate while the Goal specification may prefer the reverse. For this reason, the Planner includes an API to change the Satisfaction formula for a particular Goal instance at runtime to accommodate local policy variation.

We require Goals to be immutable, as the semantics of a Goal are built into Technique code and changes to the Goal specification will render Techniques obsolete. Consequently, evolution of a Goal's specification requires that a new specification with a new Goal name 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 some updates could be automated in certain cases.

3.2. Technique sub-Goal Search and Selection

The planning process of building and evaluating Goal Trees discussed in Section 2.3 is essentially a search through a hierarchical Technique space for "paths" through the Goal Tree that best satisfy the top-level Goal. The Planner, by default, uses a heuristic search algorithm where each Goal is independently bound to the Technique with the highest Satisfaction value, until forced by a subsequent failure, or other event, to expand the search. If two or more sub-Goals are incompatible, eval stages in Techniques call fail() to signal to the Planner that the current set of sub-Goals is unacceptable. For example, the RTPStreams Technique of Listing 1 calls fail() on line 12 if the source and sink do not support mutually acceptable stream formats. fail() terminates exploration of the corresponding subtree.

Often this declaration of failure is too radical. 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 Listing 1 suffers from a combination of deficiencies: (1) that the heuristic choice of sub-Goals does not reflect critical dependencies between sub-Goals; and (2) that a single bad combination of sub-Goal choices will occlude the exploration of lower-rated but potentially viable solutions using this Technique. The following paragraphs describe mechanism for guiding the search breadth.

```
1  to PlayMovie(name, language): via RTPStreams2:
2
3      subgoals:
4          source = RTPAVSource(goal.name, goal.language,
5                               stream_format="H.264")
6          sink = RTPAVSink(stream_format="H.264")
```

Listing 3. Goal parameters passed down the Goal Tree limit sub-Goals to H.264-compatible streams only.

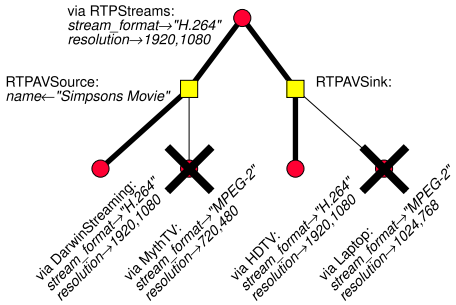

Figure 2. The RTPStreams Technique sets the stream_format parameter to H.264, causing both MPEG-2 Techniques to fail.

3.2.1. Search-narrowing Goal parameters. Instead of checking for mis-matched parameters after the fact, one simple alternative involves making decisions high in the Goal Tree and passing search-narrowing parameters down the tree for each subgoal. Listing 3 sketches a revised search for a source and sink, each specifying H.264 as the media format (restricting solutions to devices that accept or emit this media type). Such Techniques lead to a Goal Tree of the form of Figure 2. If multiple formats are to be explored, this approach requires that an alternative node be established for each plausible format combination, each requiring a separate Technique.

3.2.2. Dependent Subgoal Binding. We may improve sub-Goal search performance by ordering sub-Goal searches. For example, we might (1) search for a source emitting an 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. The attributes of sub-Goal bindings from earlier stages may be used to direct searches in subsequent ones. Listing 4 illustrates the use of this mechanism to constrain the search of our example. Figure 3 shows the resulting Goal Tree.

```
1  to PlayMovie(name, language): via RTPStreams3:
2
3      subgoals:
4          source = RTPAVSource(goal.name, goal.language)
5      subgoals:
6          sink = RTPAVSink(
7              stream_format=subgoals.source.stream_format
8              )
```

Listing 4. The second subgoals stage can use Properties from the first to guide the Planner's Technique search.

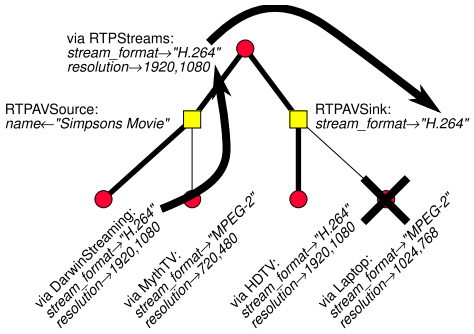

Figure 3. The RTPStreamsTechnique passes the stream_format Property of its source sub-Goal as a parameter to its sink sub-Goal, causing the Laptop Technique to fail and forcing selection of the HDTV.

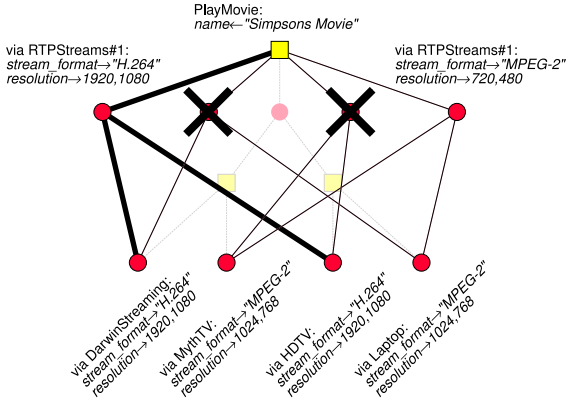

Figure 4. Goal Tree with cloned Techniques. The RTPStreams Technique is cloned for each combination of source and sink. The two pairs with matching stream formats succeed while the other two fail.

3.2.3. Tree Search and Exploration. Alternatively, the Planner may alter the search for an acceptable set of Goal/Technique bindings from the default single-pass heuristic to exhaustive exploration of all possible choices. Full search involves testing each combination of sub-Goal bindings. However, instead of testing each combination sequentially, the Planner uses a Goal Tree manipulation called *Node Cloning* to search all sub-Goal combinations while maintaining a record of the Satisfactions of each combination. Node Cloning allows the Planner to revisit particular sub-Goal combinations as the computing environment changes without running through the entire sequence of combinations. During Node Cloning, the Planner makes a copy of a Technique node but binds its sub-Goals to particular Techniques rather than to an open-ended Goal node. Figure 4 illustrates Node Cloning with the original RTPStreams Technique from Listing 1. The RTPStreams Technique is cloned four times, once for each combination of its sub-Goal bindings. Two choices have matched stream_format parameters that allow their clones to succeed; the other two clones fail.

Of course, the Planner's overuse of Node Cloning may lead to a worst-case exponential explosion in the number of choices, so complete combinatorial search is only feasible for small Goal Trees. For large Goal Trees, the Planner only clones small, heuristically chosen sub-Trees, increasing the number of choices available without affecting running time adversely.

3.3. Runtime Adaptivity

Techniques are sequential scripts, yet they need to respond to changes in Properties of sub-Goals forced by the environment and changes in Goal parameters forced by the top-level application. For example, a video Technique must respond to requests for new titles as well as bitrate changes of its sub-Goals. A naïve apparoach to this problem is to simply re-run the Technique from its first stage; however, this has performance implications for Techniques with a large number of stages or stages that must access the network.

Instead, the Planner keeps track of what Goal parameters and sub-Goal Properties each stage of each Technique uses and *rolls-back* the Technique only as far as it needs to account for changes in these tracked variables. In order to implement roll-back, the Planner saves the pre-execution state of each stage in the Technique. When a tracked variable changes, the Planner finds the first stage that depends on the variable and resets the state of the Technique to whatever pre-execution state was associated with that stage. Thus reset, the Planner re-runs the stage and any subsequent stages. For example, in Listing 1, a change to the sink's resolution property will only re-run the Technique starting at line 13. We find this roll-back strategy saves computation and network traffic and contributes to the

```
def monitor_entity_loop(tf,      # Technique Factory
                        type):  # extra arg passed by sub-Goal

    finder = find_entities_of_type(type)
    known_resources = {}
    # now update as things change
    while True:
        event = finder.get_event()
        if (event.type == 'new_device'):
            vteq = tf.new_teq()
            known_resources[event.resource] = vteq
            vteq.resource = resource
            vteq.resolution = resource.resolution
            vteq.liveness = 'alive'
            vteq.notify()
        elif (event.type == 'dead_device')
            vteq = known_resources[event.resource]
            vteq.liveness = 'dead'
            vteq.fail()
        else:
            pass

to FindRTPSink(stream_format): via VLCHost:

    subgoals:
        vlc_host = TechniqueFactory(code=monitor_entity_loop,
                                    type='VLCHost')

    eval:
        if subgoals.vlc_host.liveness != 'alive':
            planner.fail("%s not alive" % subgoals.vlc_host)
    ...
```

Listing 5. The monitor_entity_loop function creates Techniques for each resource it finds.

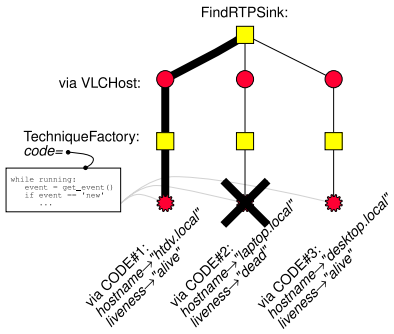

Figure 5. The VLCHost Technique uses the TechniqueFactory to call discovery system code. The Planner clones the VLCHost Technique for each Technique the TechniqueFactory's code creates. The laptop represented by CODE#2 has disappeared, causing the created Technique to fail.

Planner's ability to quickly switch among different Plans in the Goal Tree.

3.4. Technique Factories and External Events

While Techniques may call arbitrary code in their eval stages, these calls are "one-shot" — their changes are not tracked by our roll-back system. However, for certain kinds of code, including resource discovery and monitoring, outside changes should propagate to the Planner and cause re-evaluation of Techniques. To handle these cases, we provide a special sub-Goal called TechniqueFactory that allows external code to create and control Techniques directly. For example, Listing 5 shows how our VLCHost Technique uses the monitor_entity_loop function to create Techniques for resources found with a long-running discovery system.

The external code has complete control over the Techniques it creates with its factory. The code may set and re-set its virtual Technique's Properties as well as fail Techniques that are no longer applicable. In contrast to "one-shot" function calls, changes to the code-based Technique Properties are tracked like normal sub-Goal Properties, invoking the Planner's Technique restart mechanism.

If the code associated with the TechniqueFactory sub-Goal creates more than one Technique, rather than choosing the best Technique, the Planner clones the TechniqueFactory sub-Goal to expose all of its Techniques to higher-level Techniques. Figure 5 illustrates how the Goal Tree changes.

4. Applications

In order to test the general applicability of Goals and Techniques, we built several applications. Some rely entirely on the Planner to make all decisions, while other use the Planner only for parts of the application that need to be adaptive.

4.1. JustPlay Audio and Video

The JustPlay Audio [7] and Video application is an adaptive media player designed to reduce the amount of configuration users must do in order to use their various A/V-capable devices. Users control the system through a simple "voice shell" application that uses speech recognition to translate voice commands into top-level Goals. These Goals are then passed to the Planner, which continually monitors for changes in the environment and user commands that alter the top-level Goal. The JustPlay system started as audio-only, but as we built a video infrastructure, we were able to add Techniques that made the Planner aware of these new capabilities.

The Techniques used as examples in this paper come from the JustPlay application because JustPlay includes many of the technical challenges we sought to solve. For example, our video selection Technique uses sequential sub-Goal binding to more clearly define what combinations of A/V streams are acceptable for

the system. We also make use of the TechniqueFactory sub-Goal to connect the Planner to discovery subsystems. In a testament to the additivity of our system, the JustPlay application has worked with two distinct discovery systems — one with a custom in-house protocol and one based on DNS-SD — with slightly different APIs and semantics. Changing between the systems just required creating new low-level "finder" Techniques (like Listing 5). Moreover, Techniques for both systems could co-exist in the same Planner process, making it easy to gradually introduce the new discovery system.

4.2. User Proxies

We have also used the Planner to maintain user-level applications in the face of changing hardware, similar to the aims of Gaia [8], [9] and Aura [10]. In particular, we tested this with a text chat application and a teleconference application. Users modified the system by adding new Techniques that better suited their desires. For this application, a trusted machine runs a network-exported Planner that maintains user Goals. As the Planner discovers client devices (standard desktops as well as PDAs), it re-configures the text chat and/or teleconference applications to use the resources of new devices: if a user turned on his PDA after leaving his office, the system would reconnect his chats and allow him to continue as though he were still at his office computer.

4.3. Other Applications

The Planner also powers a few other applications outside of the pervasive computing field. Our crisis management application simulates several crises affecting a small city. The crisis management application benefits from the open-ended nature of the Planner because it allows new strategies to be added to the system as crises unfold. We also implemented a Recipe application that uses the Planner to choose among recipes depending on (1) what ingredients are available and on (2) the user's food preferences. Each recipe is written as a Technique, with sub-Goals for each utensil, appliance, and ingredient that the recipe requires. A GUI allows users to alter the Planner's choices by explicitly rejecting certain ingredients (which cause Techniques depending on those ingredient sub-Goals to fail) or by altering the Satisfaction formula to favor certain Properties (like caloric content, flavor, or cooking time).

Finally, we are currently exploring the use of the Planner in a hardware design "compiler". We use Goals to represent each class of hardware element, such as an adder or register, and Techniques to provide ways of stitching the circuits together. The final output is Verilog code. The Planner allows us to explore alternative implementations of the same circuit, test their performance in simulation, and more rapidly iterate hardware designs.

5. Implementation and Performance Evaluation

The Planner is written in pure Python and has been tested on GNU/Linux, Apple os x, and Microsoft Windows (under cygwin).[1] The Planner can run as both a stand-alone command-line tool as well as be linked into traditional applications. Applications that import the Planner have access to an extended API that lets the application more finely control the execution of the Planner, as well as integrate the Planner's evaluation loop with its own mainloop and/or threads.

5.1. Latency Evaluation

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; the rest of this section details the user-visible latency that the Planner adds to the application.

All tests were run on a desktop Pentium 4/3.2GHz with 1 GiB of RAM running GNU/Linux 2.6.22 and Python 2.5.1.

Our start-up latency test measures how long it takes the Planner to build, evaluate, and execute trees of various sizes. 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.

5.2. Results

Figure 6 summarizes our latency tests. Overall, startup latency scales linearly with the number of nodes in the Goal Tree. On normal size trees (≈50 nodes),

1. 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.

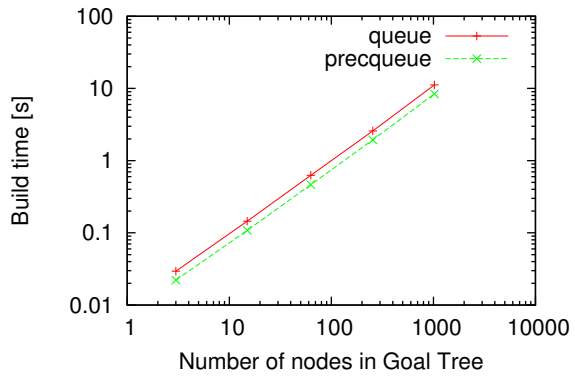

Figure 6. The latency to fully build and explore the Goal Tree as a function of the total number of nodes. The precqueue scheduler is about 14% faster than the normal queue scheduler.

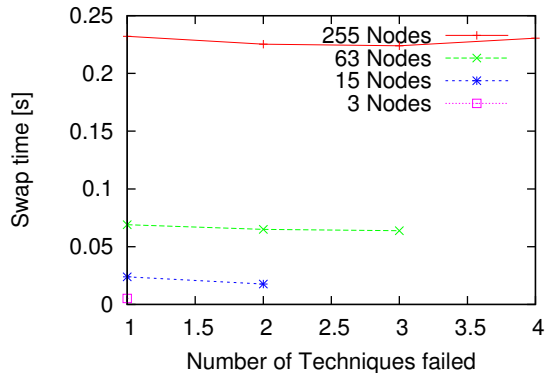

Figure 7. Goal Tree swap latency, separated by the size of the tree. The x-axis is the number of Techniques we failed in order to induce the change.

the Planner adds 1 s of startup latency. In real-world applications, we find that the Planner's startup latency is dwarfed by service discovery latencies.

In early tests, we found that performance of the Planner is tied to the order in which the Planner runs the evaluation stages of Techniques. We developed two stage schedulers: one that maintained a simple work queue (called the queue scheduler) and another, the precqueue scheduler, that would only schedule Techniques higher in the tree after lower-level Techniques finished evaluation. The precqueue scheduler was usually about 14% faster.

Figure 7 shows how long the Planner takes to react to changes in its environment and swap in new Techniques, using our precqueue scheduler. 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. We found that even a large, 255-node tree could be updated in less than 250 ms. We attribute this fast speed to our Goal Tree cache and roll-back mechanism.

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 [1], Service-oriented Network Sockets [2], and Lightweight Adaptive Network Sockets [3]. These systems allow applications to opportunistically connect to the best resources in a given environment and leave adaptation to the application.

Other frameworks provide high-level abstractions. CMU's Aura system [10] uses *tasks* to capture user-level intent. Aura uses tasks to map user intent to available resources without requiring user interaction and to optimize resource allocation according to user-specified QoS parameters. Similarly, UIUC's Gaia [8] provides event and context services for managing applications in "ActiveSpaces". We concentrate on the lower level of composing applications once context and intent have been discovered. Olympus [9] extends Gaia with a programming model for writing code portable between ActiveSpaces. Our system and Olympus solve slightly different problems. Olympus maps abstract descriptions to ActiveSpace entities using hierarchically defined ontologies in order to avoid the tedium of linking entities manually (as is required by Gaia). Goals, on the other hand, are a generic programming construct aimed at allowing open-ended decision making about any component, algorithm, or resource a pervasive application may need.

Semi-automatic service composition systems such as NinjaPaths [11] or SWORD [12] complement our approach. Such systems can be used to generate Techniques and aid in rapid development of Technique-based applications.

Declarative and implicative programming approaches, especially rule-based systems [13] and event-condition-action (ECA) systems, provide programming constructs at levels of abstraction similar to our system. For example, InterPlay [14] uses a derivative of the Jess rule system [15] to provide a pseudo-English user interface to a consumer electronics environment. The scope of InterPlay is different than our work — it does not target adaptive applications, but rather concentrates on ease of use. Our work may benefit from the UI innovations of InterPlay.

SOCAM [16] and Chisel [17] are ECA frameworks for managing events in context-aware applications. ECA systems are designed to react to changes in the environment or context while our system aims to evaluate available choices in the environment to fulfil abstract requirements. ECA systems may provide an alternative user interface to invoking Goals, e.g. "when I arrive at home, invoke PlayMusic(genre=Jazz)".

7. Conclusion

Emerging computing environments require new abstractions that permit increased levels of runtime adaptivity while still maintaining extensibility. Goals and Techniques meet both requirements by providing a structured way of decomposing adaptive applications. Goals represent open-ended choice points that can be compared by an application-generic Planner. Techniques provide specially prepared code modules that embody domain-specific knowledge. Our system allows hierarchical decomposition of applications through Technique-declared sub-Goals, but unlike competing systems, provides programmers a framework for declaring dependencies between sub-Goals. Our system is additive: new Techniques can be added without requiring changes in existing Techniques. In order to evaluate our system, we implemented four widely-varying applications using Goals and Techniques. In performance testing, we found that our system adds only a small amount of latency to application start-up and fail-over.

We are currently working on ways for users to customize the Planning process without needing to resort to programming. We are approaching this problem in two ways. First, we are working on a Satisfaction plug-in that lets users customize the Satisfaction calculation. Second, we are implementing a GUI "sand box" where users can explicitly reject choices the Planner has made, providing an easy-to-use interface to device configuration.

8. Acknowledgements

This research is sponsored by the T-Party Project, a joint research program between MIT and Quanta Computer Inc., Taiwan.

References

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley, "The design and implementation of an intentional naming system," in *SOSP*, 1999, pp. 186–201.

[2] U. Saif and J. Mazzola Paluska, "Service-oriented network sockets," in *MobiSys 2003*, 2003.

[3] U. Saif, J. Mazzola Paluska, and V. P. Chauhan, "Practical experience with adaptive service access," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 9, no. 1, pp. 27–40, 2005.

[4] C. Becker, M. Handte, G. Schiele, and K. Rothermel, "PCOM - a component system for pervasive computing." in *PerCom '04*. IEEE Computer Society, 2004, pp. 67–76.

[5] P. Costa, G. Coulson, R. Gold, M. Lad, C. Mascolo, L. Mottola, G. P. Picco, T. Sivaharan, N. Weerasinghe, and S. Zachariadis, "The runes middleware for networked embedded systems and its application in a disaster management scenario," in *PerCom*. IEEE Computer Society, 2007, pp. 69–78.

[6] V. Poladian, S. Butler, M. Shaw, and D. Garlan, "Time is not money: The case for multi-dimensional accounting in value-based software engineering," in *Fifth Workshop on Economics-Driven Software Engineering Research (EDSER-5)*, May 2003.

[7] J. Mazzola Paluska, H. Pham, U. Saif, C. Terman, and S. Ward, "Reducing configuration overhead with goal-oriented programming." in *PerCom Workshops*. IEEE Computer Society, 2006, pp. 596–599.

[8] M. Roman, 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.

[9] 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*. IEEE Computer Society, 2005, pp. 7–16.

[10] 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.

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

[12] S. R. Ponnekanti and A. Fox, "Sword: A developer toolkit for building composite web services," in *The Eleventh World Wide Web Conference (Web Engineering Track)*, 2002.

[13] NASA, "Clips reference manual," in *NASA Technology Branch, 1993*, October 1993.

[14] A. Messer, A. Kunjithapatham, M. Sheshagiri, H. Song, P. Kumar, P. Nguyen, and K. H. Yi, "Interplay: A middleware for seamless device integration and task orchestration in a networked home," in *PerCom*. IEEE Computer Society, 2006, pp. 296–307.

[15] Sandia National Laboratories, "Jess rule-based system," in *Jess User Manual*, 2003.

[16] T. Gu, H. K. Pung, and D. Zhang, "A service-oriented middleware for building context-aware services," *J. Network and Computer Applications*, vol. 28, no. 1, pp. 1–18, 2005.

[17] J. Keeney and V. Cahill, "Chisel: a policy-driven, context-aware, dynamic adaptation framework," in *POLICY 2003.*, 4-6 June 2003, pp. 3–14.