Structured Decomposition of Adaptive Applications

by

Justin Mazzola Paluska

S.B. Physics
Massachusetts Institute of Technology (2003)

S.B. Electrical Engineering and Computer Science
Massachusetts Institute of Technology (2003)

M.Eng. Electrical Engineering and Computer Science
Massachusetts Institute of Technology (2004)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Engineer in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2012

Author . . . .
Department of Electrical Engineering and Computer Science
January 18, 2012

Certified by . . . .
Steve Ward
Professor
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Professor
Chair, Department Committee on Graduate Students

Structured Decomposition of Adaptive Applications

by

Justin Mazzola Paluska

Submitted to the Department of Electrical Engineering and Computer Science
on January 18, 2012, in partial fulfillment of the
requirements for the degree of
Engineer in Computer Science

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.

Thesis Supervisor: Steve Ward
Title: Professor

Acknowledgments

No one person works or matures without the influence of others. With this in mind, I would like thank my advisor, Steve Ward, for his advice and support during my continuing journey through graduate school. Steve's many engineering and design insights helped shape and greatly improve the Planner project as well as my own abilities to design and implement systems. I deeply appreciate all Steve has done for me.

I am indebted to Hubert Pham for his continual help not only with the Planner project, but also for being a close friend always willing to hear and help resolve complaints and rants.

The Planner project is the result of a collaboration with many current and past O_2S group members—Grace Chau, Umar Saif, Chris Stawarz, Chris Terman, Jason Waterman, Eugene Weinstein, Victor Williamson, and Fan Yang. I thank each of them for their input and expertise.

I thank my family for being my foundation and for their emotional support. To my mom, thank you for teaching me to love learning and providing me with tools to succeed in life. To my sister, thank you for showing me how to persevere though tough situations. To my grandfather, thank you for always encouraging me.

Finally, to Trisha, my partner in life, I thank you for always being by my side and helping me grow as a person.

Portions of this thesis were previously published at IEEE PerCom [13] and in *Pervasive and Mobile Computing* [14].

Contents

List of Figures

List of Listings

List of Tables

Chapter 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 [8, 20, 21] 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 [2] 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 thesis, 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, *Technique* scripts describe 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.

Chapter 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. A Property is a simple key-value pair that describes a quality of the implementation the Technique provides for 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 to satisfy the Goal.

```
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 2.1: A Technique that satisfies the PlayMovie Goal by linking an RTP source stream to an RTP output device.

2.2 Techniques

A Technique is a small script mixing declarative and arbitrary imperative code broken up into a series 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 2.1 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,

```
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 2.2: Application code from a movie player that uses the Planner.

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) according to 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 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 2.2 shows typical application code while Figure 2.1 illustrates how the Planner expands the PlayMovie Goal.

In Listing 2.2, 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. Once a suitable way to satisfy the Goal is found, the application calls commit() to execute the chosen Techniques. Finally, the application monitors, and potentially updates, the running set of Techniques.

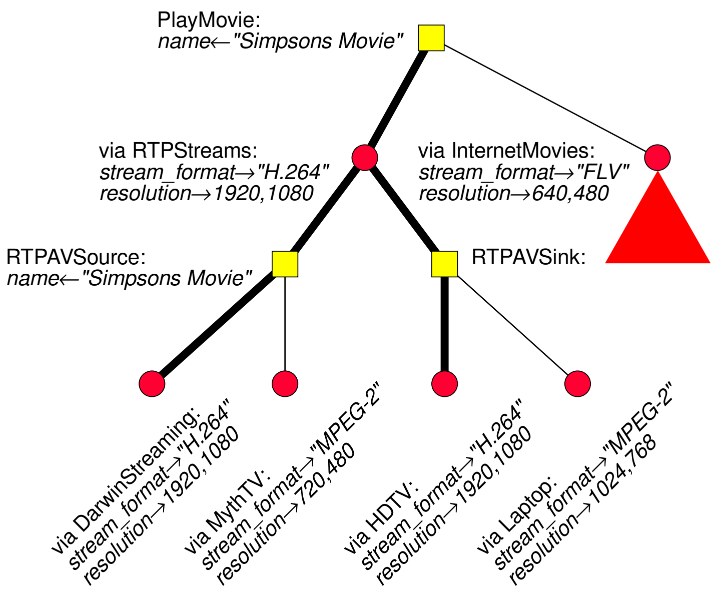


Figure 2.1: A partially explored Goal Tree for the PlayMovie Goal. Yellow boxes are Goals and show the Goal parameters. Red circles are Techniques and are displayed with their exported Properties. Thick lines represent the chosen Plan for the PlayMovie Goal.

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 2.1 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 2.1, 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.

Chapter 3

Architecture

Our system enables (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 [16] among alternative Techniques addressing each Goal. The Planner need only choose the Technique with the highest Satisfaction score at each Goal decision point.

We require Goals to be immutable, as the semantics of a Goal are built into Technique

```
((teq.screen_size / MAXIMUM_SCREEN_SIZE) and
 (teq.location == goal.location))
```

Listing 3.1: A default Goal Satisfaction formula for the FindVideoSink Goal. teq is the Technique under evaluation and goal provides access to the Goal parameters.

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.

3.1.1 Sources of Satisfaction

Our projection of all Properties of a Technique onto a single scalar has been the most controversial of our architectural decisions. At a very high level, the role of Satisfaction in our system is analogous to that of money in an economy: it dramatically simplifies decisions by reducing dimensionality of the parameter space. It can be argued that if a discrete choice is to be made between N competing alternatives, at some point the decision process must reduce all of the inputs on the N choices down to a scalar—indeed, to a small integer reflecting the choice.

The controversy, of course, revolves around the point at which the dimensionality reduction occurs. In our system, the *only* module containing code specific to a Technique is the Technique itself, making it the responsibility of each Technique to evaluate its own Properties. In order to provide some consistency in the evaluation of disparate Techniques addressing the same Goal, each Goal specification provides a default formula for computing Satisfaction for a Technique from Properties it reports; thus the semantics of a Goal imply, among other things, the way in which various cost and performance parameters of proposed Techniques effect the actual selection of an approach to be used.

Listing 3.1 illustrates a simple default policy for the FindVideoSink Goal, optimizing for a nearby large screen. The Satisfaction formula may reflect information from a variety of sources:

- Goal parameters passed from superior nodes (whose subgoals are being addressed by this Goal instance). These can specify parameters of the Satisfaction, e.g. the relative weights of cost versus performance characteristics, as well as limit the ranges of valid Properties.

- Properties returned by child Techniques, each of which reflects the way in which the Techniques will satisfy the Goal.

- External inputs. For example, local policy may dictate a different Satisfaction scheme than the default Goal policy—e.g., users may prefer higher resolution video at a lower frame rate while the Goal specification may prefer the reverse. Thus, Satisfaction values may reflect parameters stored in a local database of user preference information, allowing user customizations to influence choices.

The Satisfaction mechanism allows Planner decisions to reflect arbitrary application-specific parameters (sucn as speed, throughput, memory size, energy use, and many others) while keeping the Planner itself application-generic. It is the responsibility of each Technique to generate a Property values used to compute its scalar Satisfaction, and the Planner simply tries to optimize the Satisfaction value for its top-level nodes.

3.1.2 Code as an evaluation mechanism

Each Technique constitutes a description of an approach to satisfying a Goal, and comprises code addressing two orthogonal issues: (a) the estimation of how satisfactory its solution is likely to be, and (b) the actual implementation of the approach it promises. Our use of arbitrary code for the latter function is uncontroversial, given our desire to accommodate arbitrary mechanism in our solutions. The decision to allow arbitrary Technique code in evaluation, in contrast, invites controversy.

The cost of allowing arbitrary code for evaluation is that, in the general case, the code can only be run; it cannot be analyzed, and the most interesting questions that might be asked (e.g. whether one Technique will be uniformly more satisfactory than another under some prescribed circumstance) are formally undecidable. The expression of each Technique's evaluation code in some more constrained form—e.g., a simple logic—might allow such analysis.

Our choice was motivated largely by the broad range of evaluation mechanisms to be explored, and the fact that code is the simplest expedient for accessing them. We have experimented with language constraints that offer interesting algorithmic advantages in the planning process. For example, a `where` Technique stage that allows SQL-like restriction of sub-Goal parameters is one such experiment.

3.1.3 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). We assume Techniques, like library procedures, to 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 Goal Tree. Each subsequent evaluation will select the same buggy Technique, detect its failure, re-evaluate the Goal 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.

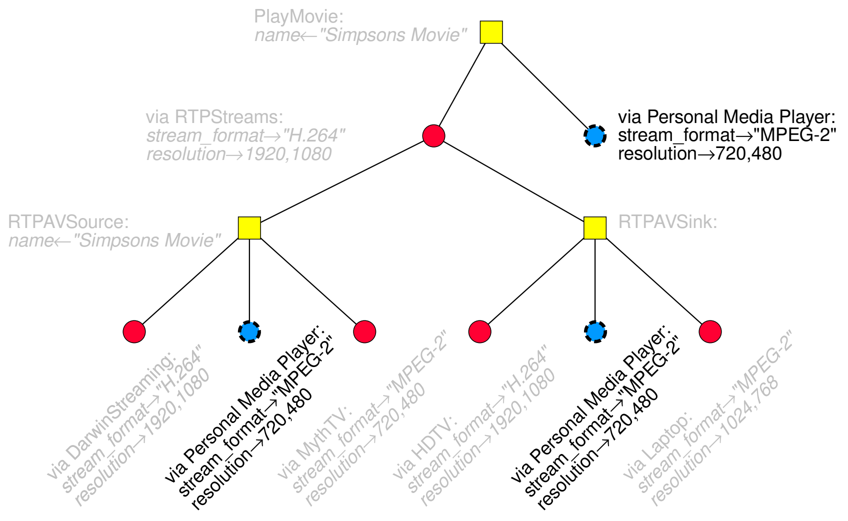


Figure 3.1: A Goal Tree incorporating a new Personal Media Player device. The media player has internal storage and a screen, so it can implement the PlayMovie Goal directly, or be used as a source or sink.

3.1.4 System Evolution

Techniques provide the foundation for long-term system evolution. Typically, we find that Techniques fall into two broad categories. The first kind, which we call *driver* Techniques, provide Technique-oriented wrappers to particular resources. Driver Techniques request notifications from discovery services to monitor the existence and health of external devices and services. In turn, these notifications are turned into Properties accessible by higher-level Techniques. In many aspects, driver Techniques are the analog of intentional device descriptions in other pervasive systems [1, 8, 20, 21]. The sample via RTPStreams Technique of Listing 2.1 represents the other kind of Technique, an *algorithmic* Technique. These Techniques are not associated with any particular resource, but rather embody a recipe for composing resources.

The Planner makes no distinction between driver and algorithmic Techniques. As such, what may be done by a single module can be replaced by a group of modules and vice versa simply by switching between Techniques.

For example, consider how a new device, such as a wifi-enabled Personal Media Player, may be used for the PlayMovie Goal. One Technique may provide an RTPAVSource interface to the device, using its network connection to stream video files stored on the player. Another Technique may provide an RTPAVSink interface, allowing the Planner to take advantage of the player's screen and headphones. Finally, a third Technique may directly implement the PlayMovie Goal using both of the Personal Media Player's built-in input and output capabilities. Figure 3.1 illustrates. The Planner may use all of the Techniques as opportunities to increase the number of alternatives available to choose from, allowing an existing Planner to evolve as the devices surrounding it evolve without modifying the Planner's core.

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 2.1 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 2.1 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.

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 3.2 sketches a revised search for a source and sink, each specifying H.264 as the media format (restricting solutions to devices that accept or

```
to PlayMovie(name, language): via RTPStreams2:

    subgoals:
        source = RTPAVSource(goal.name, goal.language,
                              stream_format="H.264")
        sink = RTPAVSink(stream_format="H.264")
```

Listing 3.2: Goal parameters passed down the Goal Tree limit sub-Goals to H.264-compatible streams only.

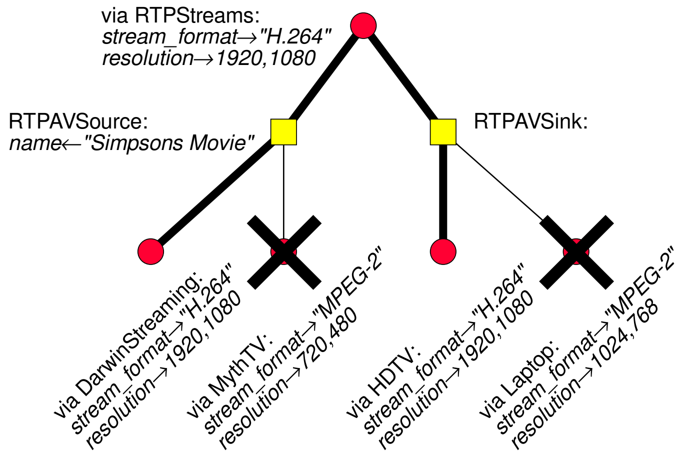


Figure 3.2: The RTPStreams Technique sets the stream_format parameter to H.264, causing both MPEG-2 Techniques to fail.

emit this media type). Such Techniques lead to a Goal Tree of the form of Figure 3.2. 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 3.3 illustrates the use of this mechanism to constrain the search of our example. Figure 3.3 shows the resulting Goal Tree.

```
to PlayMovie(name, language): via RTPStreams3:

    subgoals:
        source = RTPAVSource(goal.name, goal.language)
    subgoals:
        sink = RTPAVSink(
            stream_format=subgoals.source.stream_format
            )
```

Listing 3.3: The second subgoals stage can use Properties from the first to guide the Planner's Technique search.

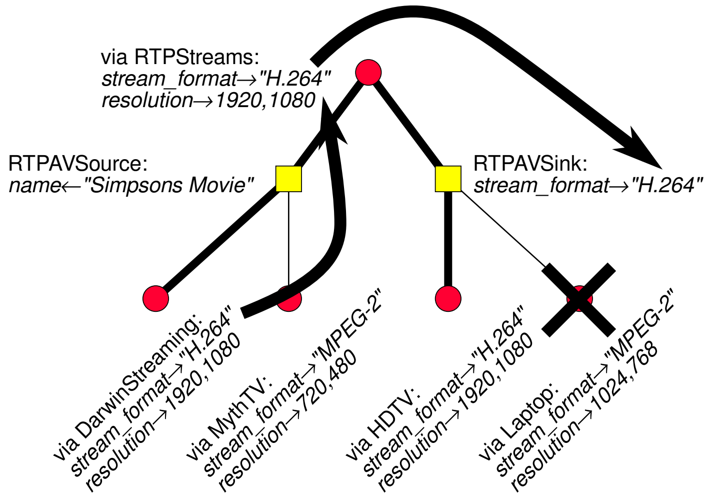


Figure 3.3: The RTPStreamsTechnique passes the stream_format Property of its source sub-Goal as a parameter to its sink sub-Goal, causing the Laptop Technique to fail and forcing selection of the HDTV.

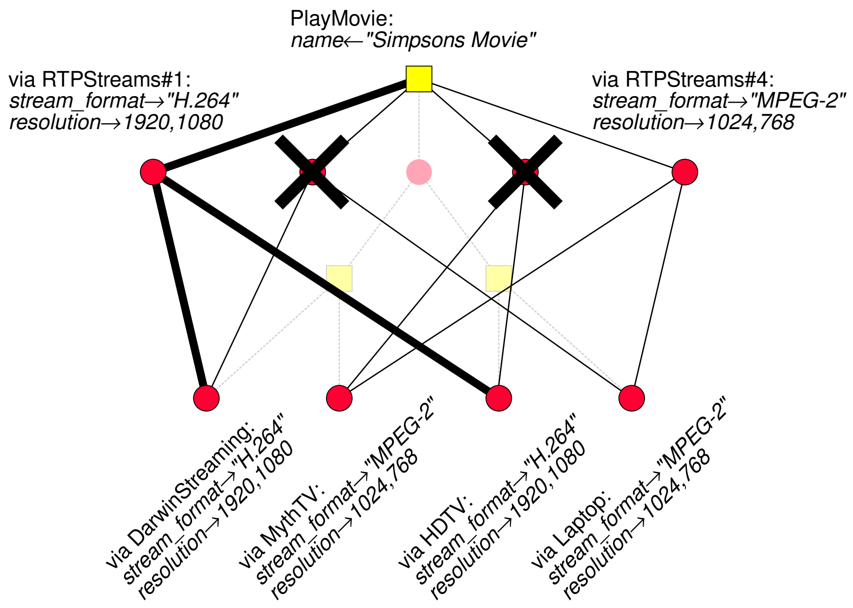


Figure 3.4: Goal Tree with cloned Techniques. The RTPStreams Technique is cloned for each combination of source and sink. The two pairs with matching stream formats succeed while the other two fail.

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. The most straightforward approach to full search is to test each combination of sub-Goal bindings sequentially and remembering what combination led to the highest Satisfaction. Unfortunately, sequential search is inefficient in the face of Property changes—e.g., if network conditions change, the Properties (and thus, Satisfactions) of only some nodes may change, yet the Planner must loop through all node combinations again.

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. In order to Node Clone, the Planner makes a copy of a Technique node and binds its sub-Goals to particular Techniques rather than to an open-ended Goal node. The newly cloned Techniques function like standard Techniques, and as such can be re-evaluated as conditions change.

For example, Figure 3.4 illustrates Node Cloning with the original RTPStreams Technique from Listing 2.1. The RTPStreams Technique is cloned four times, once for each

```
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)
     ...
```

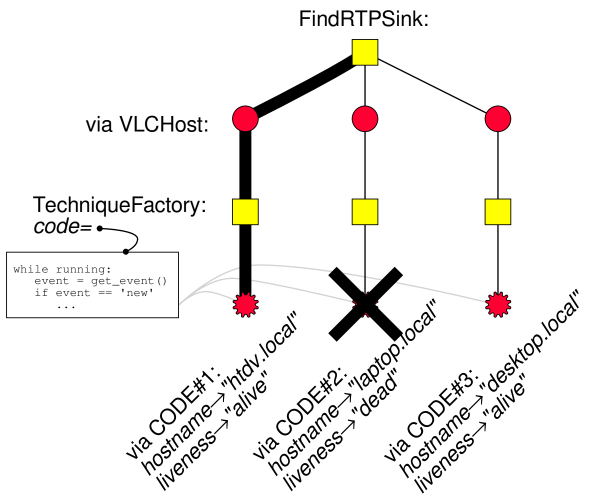
Listing 3.4: The monitor_entity_loop function creates Techniques for each resource it finds.

combination of its sub-Goal bindings. Two choices have matched stream_format parameters that allow their clones to succeed; the other two clones fail. Node Cloning interacts well with our Technique re-evaluation system: e.g., if the via MythTV Technique changes its properties, we must only revisit the two clones that depend on it.

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 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


FindRTPSink:
via VLCHost:
TechniqueFactory:
code=
via CODE#1: hostname→"htdv.local" liveness→"alive"
via CODE#2: hostname→"laptop.local" liveness→"dead"
via CODE#3: hostname→"desktop.local" liveness→"alive"

```
while running:
   event = get_event()
   if event == 'new'
      ...
```

Figure 3.5: 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.

code, such as 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 3.4 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 3.5 illustrates how the Goal Tree changes.

3.4 Efficient Runtime Adaptivity

Techniques are sequential scripts, yet they need to respond to changes in Properties of sub-Goals forced by the environment (as outlined in the previous section) 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 approach 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 2.1, 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 Planner's ability to quickly switch among different Plans in the Goal Tree.

Chapter 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 [10] 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, such as PlayMusic(artist="Beatles") or PlayMovie(name="Simpsons Movie"). 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 extend JustPlay to handle video by adding new Techniques that made the Planner aware of the new video playing 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 sub-systems. 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 "driver" Techniques (like Listing 3.4). 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 [18, 19] and Aura [6]. In particular, we tested this with a text chat application and a teleconference application. For this application, a trusted machine runs a network-exported Planner that maintains user Goals, each satisfied by custom Techniques that implement the user's desired system behaviors. Users modify the system by adding new Techniques that better suit their desires.

In a typical example, a mobile-based chat client was used in "follow-me" mode during a conversation that continued as one peripatetic participant wandered from one room to the next. Tracking that user's physical location, the system uses Goal-oriented planning in a best-effort attempt to satisfy its top-level Chat Goal using the most appropriate resources available in each room, for example switching the incoming video stream from the tiny display of the user's handheld to a communal wall-mounted plasma display wherever feasible. One Technique that implements the Chat Goal registers with a service discovery system to find all hosts owned by the user. As the Planner discovers client devices (standard desktops as well as PDAs), it invokes a ChatManager(host=...) Goal. A Technique satisfying ChatManager invokes sub-Goals TextInput(host=...) and GUI(host=...) to obtain device-specific input and output mechanisms.

For example, if the user is using his desktop, he might see a full GUI with audio-conferencing support. However, if the user turns on his PDA and leaves his office, the system will reconnect his chats by invoking the ChatManager(host="pda") Goal and allow him to continue his chats using a PDA-optimized interface.

4.3 Hardware Design

Outside of pervasive computing applications, we have used the Planner and goal-oriented programming in a circuit design system called Fide [4]. Hardware designers must make many trade-offs in the course of building a circuit. For example, one particular design may be fast, but use too many gates, while another may meet functional requirements, but only on a particular kind of hardware substrate. Fide enables a hardware designer to efficiently explore trade-offs in the circuit design process.

Fide uses Goals to represent functional requirements, from low-level requirements like a single-bit reigster to higher-level abstractions like a 32-bit adder. Techniques provide competing implementations—such as a simple ripple-carry adder, a faster carry-select adder, or even a "native" adder built-in to a particular FPGA. Fide requires each Technique to export two Properties: the estimated size of the circuit that Technique represents and an estimate of the cycle time of the circuit. The default Satisfaction formula considers both the size and speed Properties equally though the designer can override the particular weighting of size and speed in a particular run.

In contrast to Planner-based pervasive applications where the Planner is infrastructure hidden from view of the user, Fide exposes the Planner and its Plan Trees to the hardware designer so that he can directly adjust Goal parameters to tweak the circuit and explicitly explore trade-offs. Doing so allows the designer to better understand the design that Fide proposes as well as gain an intuition for different implementation strategies.

The final output of Fide is a hardware description in a language like Verilog or JSim. The quality of the circuits produced by Fide is comparable to hand-optimized circuits [4].

4.4 MusicPlanner

Yang's MusicPlanner [25] uses the Planner as the central component of a music recommendation and exploration system. The MusicPlanner enables a user to find songs similar to songs that he already enjoys and then play those songs from an extensible set of music repositories.

The MusicPlanner uses two top-level Goals: the Recommend Goal, to find recommended songs given a seed song, and the Play Goal, to play a specific song. The Recommend Goal is satisfied by a set of Techniques the each encompasses a particular way of recommending songs. Yang implements strategies that use (1) computed similarity scores or (2) overlap in user-generated "tags" to find songs comparable to the original seed song. The Play Goal works similarly to the PlayMusic Goal of JustPlay.

The MusicPlanner uses a sophisticated Satisfaction metric that takes into account user preferences as well as an abstract music similarity score to choose the best Recommend Technique. A small user study shows that users prefer the playlists generated by the MusicPlanner when it used the Planner to choose the best Recommend Technique for each song in the playlist compared to the playlists the MusicPlanner generates when it used only one recommendation algorithm.

4.5 Web Search

Williamson uses the Planner as a platform for multi-faceted and extensible web search [24]. Williamson's system defines a few top-level Goals for searching different kinds of materials on the web. Techniques satisfy those Goals using particular web services like YouTube, Google, and Yahoo.

A notable aspect of Williamson's system is that it allows users to upload their own Goals and Techniques that the Planner can immediately use. There are two consequences that flow from this feature. First, the search system must be security hardened. Since, as discussed in Section 3.1.3, the Planner treats all Goals and Techniques as trusted code, if the Planner is to run arbitrary user-uploaded code, the Planner must be sandboxed to prevent malicious users from harming the search system. Williamson elaborates on the threat model the Planner faces, though he does not fully sandbox the Planner.

Second, users must be able to debug their Techniques. Like Fide, Williamson's search system exposes the Planner's Goal Tree to the user so that he may inspect the Techniques he uploaded and amend them as necessary.

4.6 Other Applications

The Planner also powers a few other small applications. 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).

Chapter 5

Implementation

The Planner is written in pure Python and has been tested on GNU/Linux, Apple os x, and Microsoft Windows (under cygwin). 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 or threads.

5.1 The Planner Scheduler

Internally, the Planner module is similar in design to single-threaded, "mainloop" applications. In mainloop architectures, the application is built around a single event loop (typically invoking the select() system call) that surveys the work that needs to be done and schedules some unit of work for execution. In each iteration of its event loop, the Planner chooses a single node from the Goal Tree and runs a single evaluation stage of that node.

The scheduling policy the Planner modules uses is pluggable. We implemented three schedulers to explore how much node scheduling matters to the planning process. Our first scheduler, the QueueScheduler maintains a ready queue of nodes that need to be re-evaluated and adds nodes that become ready to the end of the queue. A second scheduler, the TreeScheduler applies the heuristic that if a node's children need to be updated, changes in their solution will affect the parent node, so the children should be updated before the parent. The TreeScheduler does not maintain a ready queue, but rather, on every iteration, performs a topological sort of the Goal Tree and returns the node lowest in the tree that needs updating. Unfortunately, topological sort is linear in tree size, so we also test a compromise scheduler, the PrecedentQueueScheduler, that maintains a ready queue, but adds new nodes to the queue before their parents. The PrecedentQueueScheduler, like the TreeScheduler also tries to avoid scheduling a node redundantly, but operates in time linear to the queue size rather than the entire tree size.

5.2 Technique Roll-back

As discussed in Section 3.4, a complication of our planning process is that a Technique may be "reverted" to a previously run evaluation stage if changes in sub-Goal Properties obsolete the calculations the stage performed. For a particular Technique, given a set of changes in subgoal Properties, the Planner must determine (1) what stage of the Technique to re-run and (2) what values must be restored to the Technique's state to completely revert it.

5.2.1 Stage Tracing

In order to determine what stages must be re-run when the environment changes, the Planner must know what variables each Technique phase uses and the values that each Technique stage read when it last executed. Rather than burdening the Technique writer with this bookkeeping, we use low-level Python mechanism to trace variable references within Technique code.

Technique execution tracing is implemented by a two-fold strategy. First, we require that all variables, such as Goal Parameters or Technique Properties, that may force re-evaluation of Techniques be stored as attributes of instances of a class called Solution. Second, in the Solution class, we override the standard Python mechanisms for retrieving, mutating, and removing attributes with code the keeps a log of how the attributes are accessed.

Every time a Technique stage successfully runs to completion, the Planner checks the log of every Solution object to which the Technique has access, and records what attributes were accessed and what attribute values were read. When a Solution attribute changes, the Planner need only look up what Techniques read that attribute, and re-run the each Technique starting at the earliest stage that depends on the attribute.

Our execution tracing strategy narrowly reflects the actual dynamic execution path through Technique code: it does not capture all the variables that a Technique stage *may* access on *every* execution. For example, if a Technique stage has an if/else conditional whose consequent and alternative access different variables, our code will only record the variables read by the branch taken. However, this narrowness is not a problem in practice. The Planner traces every variable the stage reads, and thus, records the variables that lead to the choice of one conditional branch over another. If some attribute change would require the Technique to execute a different conditional branch, then the Planner must have recorded the variables used to make the branch decision. Therefore, the change will cause the stage containing the branch decision to be re-run, forcing the Technique to re-execute using the new branch of the conditional.

5.2.2 State Restoration

In order to revert Technique state when we roll-back a Technique, the Planner requires that all Techniques store their state as Properties. Before an evaluation stage is run, the Planner takes a snapshot of the Technique's Properties as the "before" snapshot for the stage. When a stage needs to be re-run, the Property snapshots for all subsequent stages are erased and the Technique's Properties are set to the "before" snapshot of the stage to be run.

Normally, Technique Properties are publicly exported and made readable to the Goals and Techniques higher in the tree, e.g., the resolution and stream_format Properties of the Techniques used as examples in this text. A Technique may not want to store its internal state as a public Property. Therefore, following Python convention [23], Techniques create *private Properties* by prefixing their Property name with an underscore. Private Properties are still restored properly by the Planner on roll-back, but are not readable outside of the Technique that created them.

5.2.3 Solutions and the TechniqueFactory sub-Goal

As outlined in Section 3.3, Techniques can access any library code available to the Planner process. Normal access to library code bypasses the Planner's tracing mechanisms and will not cause Technique roll-back. When rollback is needed, the Technique must use the TechniqueFactory sub-Goal.

Our implementation of the TechniqueFactory sub-Goal spawns a thread that runs the TechniqueFactory's code argument. Each time the code creates a new virtual Technique using the new_teq() method, the Planner creates a new Solution object to store the Properties of the virtual Technique. The Solution objects of the virtual Techniques are traced like the Solution objects of normal Techniques, allowing external code changes to cause roll-back.

Chapter 6

Performance Evaluation

We perform both micro-benchmarks to show the overheads inherent to our approach, as well as macro-benchmarks on the JustPlay application to show how our system performs in typical situations. 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.

6.1 Micro-benchmarks: 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.

6.1.1 Experimental Setup

We generated a set of stub Techniques that induce large Goal Trees. 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 5. The Techniques contain no evaluation or commit code, so the measured execution time of the Planner is solely due to Planner overhead.

Our start-up latency benchmark measures how long it takes the Planner to build, evaluate, and execute trees of various sizes. After the Planner has converged on a particular Goal Tree and idled, we introduce a variety of Property changes to the leaf nodes to

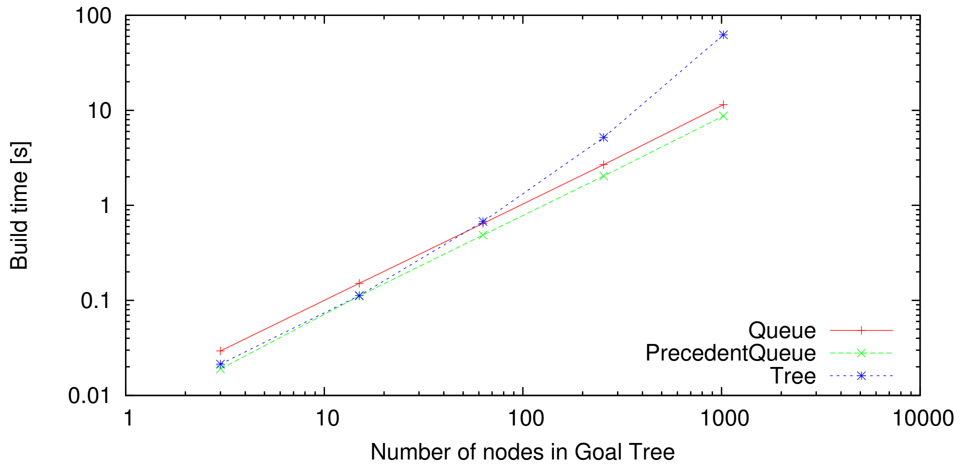


Figure 6.1: The latency to fully build and explore the Goal Tree as a function of the total number of nodes.

simulate failures. Our swap latency benchmark measures how long the Planner takes to converge to a new plan after the failures are introduced.

We run each benchmark 10 times and report the mean statistic of the 10 runs.

6.1.2 Results

Figure 6.1 summarizes our latency tests. For queue-based schedulers, startup latency scales linearly with the number of nodes in the Goal Tree. On normal size trees (≈50 nodes), 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 (see section 6.2).

We find that performance of the Planner is tied to the order in which the Planner runs the evaluation stages of Techniques, and thus, the scheduler that the Planner users. As Figure 6.1 shows, the Planner's start-up latency can be reduced with a smart scheduler that reduces the amount of work the Planner must do. On small trees, the QueueScheduler has the worst performance by a factor of two. This is primarily because the QueueScheduler takes approximately twice as many iterations to converge to a stable solution than the other schedulers (see Figure 6.2). The result makes sense because each node gets scheduled every single time a child node updates itself, leading to repeated work. The PrecedentQueueScheduler and TreeScheduler require approximately the same number of iterations to explore the tree because they both avoid repeating work at higher levels in the tree until lower levels have stabilized. Unfortunately, the TreeScheduler's performance does not scale

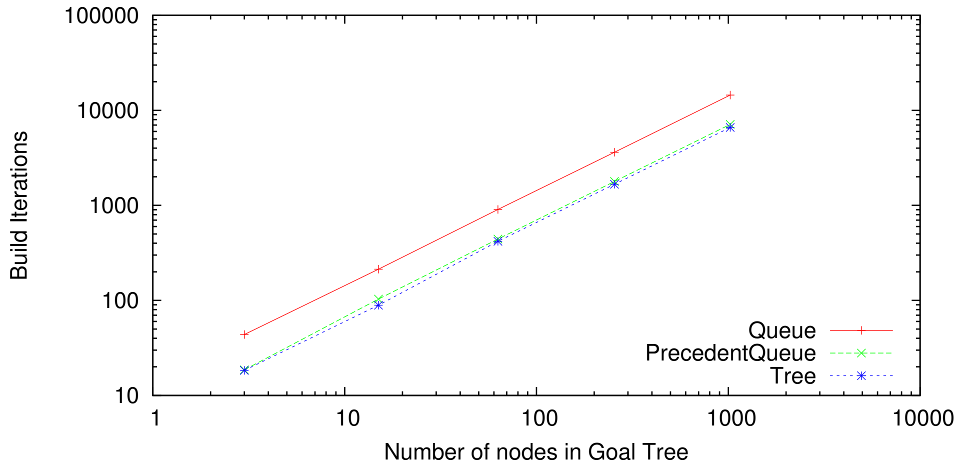


Figure 6.2: The number of scheduler iterations fully build and explore the Goal Tree as a function of the total number of nodes.

with the size of the tree because it must perform a linear-time topological sort of the tree on each iteration.

Overall, the PrecedentQueueScheduler has a 14% speed advantage over the best times of the other schedules because its time complexity is linear in ready queue size and the ready queue of nodes is usually much smaller than the entire tree. All of our schedulers completely explore each Goal Tree. Future schedulers may completely avoid working on certain Goal Tree branches, such as those with low Satisfaction scores—further reducing work and increasing performance.

Figure 6.3 shows how long the Planner takes to react to changes in its environment and swap in new Techniques. The swap test 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. Using either of the queue-based schedulers, we found that even a large, 255-node tree could be updated in less than 250 ms. The TreeScheduler performs poorly for larger trees, again, due to the topological sort on each iteration.

6.2 Macro-benchmarks: JustPlay

For our macro-benchmarks, we invoked the PlayVideo Goal using the Techniques from the JustPlay application described in Section 4.1. In the environment, we placed a single RTSP video server as well as two RTSP video sinks, leading to a Goal Tree with 18 nodes. For

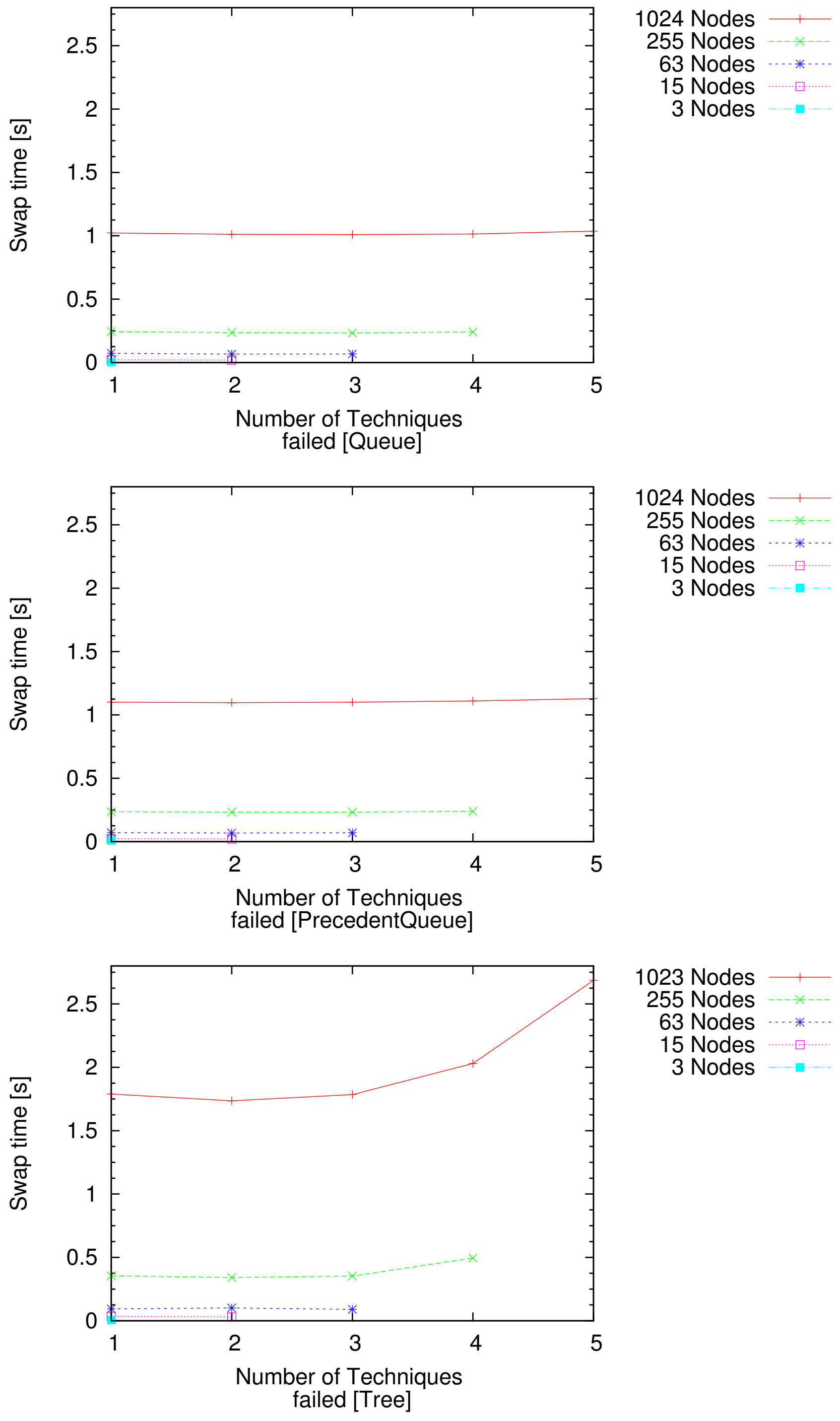


Figure 6.3: Goal Tree swap latency, separated by the size of the tree for each of the three schedulers. The x-axis is the number of Techniques we failed in order to induce the change.

Phase	Runtime [s]
Explore	0.154
Commit	0.464
Startup Sub-Total	**0.617**
Re-explore	0.038
Update	0.631
Hotswap Sub-Total	**0.669**

Table 6.1: Mean runtimes for four phases of the JustPlay application.

service discovery and RPC, we used an in-house system called NPOP [15].

6.2.1 Experimental Setup

We measure the execution times for four stages of the JustPlay application. First, "Explore" measure how long the Planner takes to construct and explore the Goal Tree for the PlayVideo Goal. Next, once a viable Plan is found, "Commit" time measures how long it takes the Planner to execute the commit stages of the JustPlay Techniques and actually start playing video. After the video plays, we introduce a new, superior Video RTSP video sink. The "Re-explore" time measures how long the Planner takes to react to the new video sink and choose a new Plan. The Re-explore time does not include the discovery latency of the service discovery system; we measure the time from when the discovery process delivers a message to the Planner process to when the Planner process chooses a new Plan. Finally, the "Update" time measures how long the Planner takes to hotswap the old sink for the new sink.

We run each benchmark 10 times and report the mean statistic of the 10 runs.

6.2.2 Results

Table 6.1 summarizes our results. The results are about a factor of 5 slower than would be predicted from our micro-benchmarks, most notably because the JustPlay benchmark includes non-empty evaluation stages and commit stages. The latency is still under a second, well within the normal startup time for a typical desktop application.

The slowdown in the evaluation and execution stages of our Techniques stems from the cost of RPC calls used to connect the video source to the video sink and start streaming, as shown by the high "Commit" and "Update" times. Note that the total hotswap runtime is higher than the cost of building a new Goal Tree from scratch. This is because the Planner must shut down the old video sink, transfer state to the new video sink, and finally initialize the new video sink. All three actions require many RPC calls over the network.

The "Re-explore" is very fast. This is because the Technique roll-back system only runs stages that need to be run, avoiding many RPCs.

Chapter 7

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 [20], and Lightweight Adaptive Network Sockets [21]. 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 [6] 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 [19] 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 [18] 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 [3] or SWORD [17] 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

[12] and event-condition-action (ECA) systems, provide programming constructs at levels of abstraction similar to our system. For example, InterPlay [11] uses a derivative of the Jess rule system [22] 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 [7] and Chisel [9] 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 fulfill abstract requirements. ECA systems may provide an alternative user interface to invoking Goals, e.g. "when I arrive at home, invoke PlayMusic(genre=Jazz)".

Chapter 8

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 sub-Goals declared by Techniques, but unlike competing systems, provides programmers with 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.

Bibliography

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

[2] Christian Becker, Marcus Handte, Gregor Schiele, and Kurt Rothermel. PCOM - a component system for pervasive computing. In *PerCom '04*, pages 67–76. IEEE Computer Society, 2004.

[3] 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.

[4] Man Ping Grace Chau. Goal-oriented hardware design. M.S. thesis, Massachusetts Institute of Technology, 2008.

[5] Paolo Costa, Geoff Coulson, Richard Gold, Manish Lad, Cecilia Mascolo, Luca Mottola, Gian Pietro Picco, Thirunavukkarasu Sivaharan, Nirmal Weerasinghe, and Stefanos Zachariadis. The runes middleware for networked embedded systems and its application in a disaster management scenario. In *PerCom*, pages 69–78. IEEE Computer Society, 2007.

[6] 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.

[7] Tao Gu, Hung Keng Pung, and Daqing Zhang. A service-oriented middleware for building context-aware services. *J. Network and Computer Applications*, 28(1):1–18, 2005.

[8] Jong Hee Kang, Matthai Philipose, and Gaetano Borriello. River: An infrastructure for context dependent, reactive communication primitives. In *WMCSA*, pages 77–. IEEE Computer Society, 2003.

[9] John Keeney and Vinny Cahill. Chisel: a policy-driven, context-aware, dynamic adaptation framework. In *POLICY 2003.*, pages 3–14, 4-6 June 2003.

[10] Justin Mazzola Paluska, Hubert Pham, Umar Saif, Chris Terman, and Steve Ward. Reducing configuration overhead with goal-oriented programming. In *PerCom Workshops*, pages 596–599. IEEE Computer Society, 2006.

[11] 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*, pages 296–307. IEEE Computer Society, 2006.

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

[13] Justin Mazzola Paluska, Hubert Pham, Umar Saif, Grace Chau, Chris Terman, and Steve Ward. Structured decomposition of adaptive applications. In *PerCom*, pages 1–10, 2008.

[14] Justin Mazzola Paluska, Hubert Pham, Umar Saif, Grace Chau, Chris Terman, and Steve Ward. Structured decomposition of adaptive applications. *Pervasive and Mobile Computing*, 4(6):791–806, 2008.

[15] Hubert Pham. A distributed object framework for pervasive computing applications. M.Eng. thesis, Massachusetts Institute of Technology, 2005.

[16] Vahe Poladian, Shawn Butler, Mary Shaw, and David 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.

[17] 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.

[18] 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*, pages 7–16. IEEE Computer Society, 2005.

[19] Manuel Roman, Christopher Hess, Renato Cerqueria, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. A middleware infrastructure for active spaces. *IEEE Pervasive Computing*, pages 74–83, October-December 2002.

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

[21] 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.

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

[23] Guido van Rossum and Barry Warsaw. Style guide for python code. `http://www.python.org/dev/peps/pep-0008/`, January 2001.

[24] Victor Lamont Williamson. Goal-oriented web search. M.Eng. thesis, Massachusetts Institute of Technology, 2010.

[25] Fan Yang. Adaptive music recommendation system. M.Eng. thesis, Massachusetts Institute of Technology, 2010.