A Goal-Driven Planning System in Java for the Oxygen Platform

by

Pavel Langer

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Computer Science
and Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

May 24, 2002

Copyright 2002 Pavel Langer. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author__________________________
Department of Electrical Engineering and Computer Science
May 24, 2002

Certified by_____________________
Steve Ward
Thesis Supervisor

Accepted by_____________________
Arthur C. Smith
Chairman, Department Committee on Graduate Theses
A Goal-Driven Planning System in Java for the Oxygen Platform
by
Pavel Langer

Submitted to the
Department of Electrical Engineering and Computer Science

May 24, 2002

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science
and Master of Engineering in Electrical Engineering and Computer Science

Abstract

The goal-driven planning system provides an extensible and distributed design for a
planning system that can be used for Project Oxygen. The extensibility is achieved
through the factoring of local planning decisions into separate modules called techniques.
The distribution is reached by connecting pieces of code working on remote machines
into a pebbles layer. The driving application for the design of the planning system was an
application which plans a teleconference between two parties by establishing audio and
video connections between them. The prototype of the teleconference application lets
users to inspect the planning process in 2D and 3D views and also step through a sample
scenario.

Thesis Supervisor: Steve Ward
Title: Professor of Electrical Engineering and Computer Science
# 1. Table of Contents

1. Table of Contents ..................................................................................................... 3
2. List of Tables ............................................................................................................ 4
3. List of Figures .......................................................................................................... 4
4. Introduction ............................................................................................................. 5
5. Background ............................................................................................................. 6
6. Planning Layer ......................................................................................................... 7
   6.1. Goal Representation ............................................................................................. 7
   6.2. Goal Trees ............................................................................................................. 12
   6.3. Satisfaction Scale ............................................................................................... 13
   6.4. Techniques ............................................................................................................. 15
   6.5. Dynamic Goal Tree Reevaluation ...................................................................... 20
   6.6. Contexts ................................................................................................................. 23
      6.6.1. Context Lookup Algorithm ......................................................................... 25
7. Pebbles Layer .......................................................................................................... 28
8. Teleconferencing Application .................................................................................... 30
   8.1. Use for Debugging ............................................................................................. 41
   8.2. Distribution through Java RMI ........................................................................... 42
      8.2.1. Serialized vs. Remote Objects .................................................................... 44
9. Suggestions for Further Work .................................................................................... 46
   9.1. Local Pebble Interaction Improvement ................................................................. 46
   9.2. Goal Tree Size Minimization .............................................................................. 46
      9.2.1. Using “Smarter” Techniques ....................................................................... 46
      9.2.2. Merging Duplicates ..................................................................................... 46
      9.2.3. Caching by Contexts .................................................................................... 47
      9.2.4. Parameterization ......................................................................................... 47
   9.3. Dynamic Libraries of Techniques ....................................................................... 48
   9.4. Improvements in Satisfaction Calculations .......................................................... 48
   9.5. Security .................................................................................................................. 49
10. Conclusion ................................................................................................................. 49
11. Appendix .................................................................................................................. 50
   11.1. Sample Techniques for the Teleconference Demo ................................................ 50
12. References ................................................................................................................. 56
2. List of Tables

Table 1. Drawbacks of passing arguments by value or by reference in a distributed system. ................................................................................................................................. 45

3. List of Figures

Figure 1. A Sample Context-free Grammar................................................................. 8
Figure 2. The structure of a goal tree. Nodes in bold is a sample execution sub-tree (only one of the planlet nodes can be selected and all of the choice nodes should be selected). ............................................................................................................ 12
Figure 3. Satisfaction scale. .................................................................................... 13
Figure 4. A sample goal tree showing the satisfaction calculation algorithm in action. Default technique for some goal means that it is the only technique designed to satisfy that goal. The best plan is in bold and choice nodes are underlined. ........ 15
Figure 5. GoalTreeNode interface ........................................................................ 23
Figure 6. Technique interface ................................................................................ 23
Figure 7. Possible object lookup within context hierarchy.................................... 27
Figure 8. Context interface ..................................................................................... 27
Figure 9. Host, Pebble, and PebbleProxy interfaces ............................................. 30
Figure 10. Teleconference demo scenario, step 1. .................................................. 32
Figure 11. The connection diagram for the teleconference demo scenario, step 1..... 35
Figure 12. 3D view of the goal tree for the teleconference demo script, step 1......... 37
Figure 13. Teleconference demo scenario, step 2. ................................................... 38
Figure 14. The connection diagram for the teleconference demo scenario, step 2.... 38
Figure 15. Teleconference demo scenario, step 3. .................................................. 39
Figure 16. The connection diagram for the teleconference demo scenario, step 3.... 39
Figure 17. The connection diagram for the teleconference demo scenario, step 4.... 40
Figure 18. Teleconference demo scenario, step 4. .................................................. 41
4. Introduction

The goal of this research was to design and implement the prototype of a goal-driven planning system, to investigate its use for a teleconferencing planning, and to build a demo, which will show major advantages of such a planning system.

Our system is intended for dynamically configurable planning based on goal resolution methodology. A large goal is broken up into smaller goals to the point where direct actions can be taken to satisfy the original goal. The main driving application for this system was a teleconferencing application, in which an audio/video connection is established between two parties. The prototype of this system was written in Java programming language, the strongly-typed syntax of which presents several challenging puzzles for an implementer requiring the use of objects whose behavior can only be determined at run time. The examples of such objects are closures (procedures with delayed evaluation) and dynamically downloaded contexts, which contain local user information including data and techniques for goal satisfaction.

The goal-driven planning system was designed in the spirit of project Oxygen’s principles [1], which mandates that computing of the future should be pervasive and embedded in all of the devices surrounding the user and it should also follow the user’s location without the need to reboot or reconfigure the system. Project Oxygen stemmed from the collaboration of MIT Laboratory for Computer Science, Artificial Intelligence Laboratory and an industry alliance including several large companies such as HP, Nokia, Phillips and others. The teleconferencing application of our goal-driven planning system satisfies Oxygen’s requirements by encompassing all of the audio/video devices available to the user such as a desktop, a handheld, and a cell phone, and also by dynamic recalculation of the communication path as the user moves about or changes the communication means.

The topic of this thesis is the prototype for the described planning system. The teleconference application is used to research and test the problem areas of our approach. Itself it lacks several real-world features that are not essential to testing the goal-driven planning platform. For example, the input goal is entered into the system in the form of a
string (basically, a keyword) rather than via voice because speech recognition is out of scope of this thesis. On the other hand, certain other features, such as 3D visualization, were included into the prototype to make debugging easier and demonstrations more appealing.

5. Background

Three areas of research influenced the design of our goal-driven planning system: project Oxygen, Prolog’s goal resolution, and planning algorithms in Artificial Intelligence.

Project Oxygen’s vision includes existence of anonymous devices that can be automatically configured based on the identity of the nearest user. In our case, these devices are microphones and cameras, or conference rooms in general, that can automatically take over a cell-phone or a handheld when a user involved in a teleconference moves closer to these stationary devices.

Programming language Prolog contains syntactic constructs to express rules based on goal resolution [8]. For example, one might write a rule that states that in order to prove the property A about object X (written as A(X)), one needs to show that X exhibits both the property B and the property C. There can be several rules similar to this one about A(X) and any one of them can be used to prove A(X). We take a similar approach in our goal-driven planning system, where we are building a goal tree consisting of alternating OR nodes and AND nodes. OR nodes are similar to having several rules for the same property (such as A(X) above), and AND nodes are similar to enumeration of dependent properties (such as B(X) and C(X) above).

Promising algorithms in AI planning based on Graphplan and SAT-compilation techniques did not prove directly applicable for our planning system [3][4]. The problem model that they are solving is somewhat similar to the teleconferencing problem, but there are two major distinctions that seem to indicate that at least direct application of Graphplan or SAT-compilation might not be appropriate. First of all, the solution that those algorithms find involves a sequence of actions spread out in time. The solution to the teleconferencing problem, on the other hand, involves a set of connections between
distributed components. This set does not exhibit a temporal aspect. Secondly, either Graphplan or SAT-compilation techniques attempt to find any solution to the presented problem. In our case, some solution can quite often be easily found. The challenge is in selecting the one with the best satisfaction value out of numerous possible solutions.

The current version of our planning system in Java owes many of its design ideas to the original implementation by Prof. Steve Ward done in Curl programming language [2].

6. Planning Layer

Our system is goal-oriented, which means that the notion of a goal is central to all the other components. Goals serve as a computer representation of some person’s intent. To simplify the human interface to computers, goals can be entered into the system through a voice shell – a program running on Oxygen hosts that listens to the user’s speech and processes it with speech recognizers and parsers to dispatch it to the correct service. For example, for the teleconference application, user Alice might press the “Talk” button on her handheld and say “Please connect me to Bob.” Voice shell running on this device would then record the speech and process it either locally or remotely by sending it to a more powerful server and transform the waveform using a speech recognizer and a parser into the goal representation, which preserves Alice’s intent.

6.1. Goal Representation

Goals can be represented within a computer in several different ways. One approach would be to make goals similar to the structure of the parsed sentence that expresses the original intent. Natural language parsers assemble a parse tree from the sentence using a (usually context-free) grammar. For example, suppose one needs to write the grammar that would parse the following sentences:

Who is this person?
Please connect me to Bob.
Send this document to the nearest printer.

One possible grammar would be the following context-free grammar:
Figure 1. A Sample Context-free Grammar.

where "→" means "expands to", "|" means "or", lower case words is the actual text that appears in the sentence, and upper case words are non-terminals, which means that they should be expanded.

Note, that this grammar can also generate many other meaningless sentences (for example, *Please please please who is the nearest printer to this person*), but it should not be a problem if there is a semantic checker in place.

The parse tree for the sentence *Please connect me to Bob* using the grammar in Figure 1 would be

```
      SENTENCE
     /-------\
   /         \
  please     SENTENCE
     |         |
   /       \
  COMMAND  OBJECT
   /     \
connect   me
```

This is the parse tree that can be used to represent the goal.

Another approach for goal representation is the idea of feature sets that is currently being developed by David Saff at MIT [5]. Feature sets can describe several concepts, goals in
particular. In the simplest form, feature sets are sets that contain a number of key-value pairs, where key is a keyword that is used to look up values, which can be arbitrary objects. Feature sets also add several other useful ideas, such as scoping of keywords to avoid clashes between them and explicit data-typing of the values to make syntactic checks easier. There will also be standard ways of conveying some metadata to the processing applications, such as to indicate that a certain keyword must be understood, i.e. if the application that processes a particular feature set encounters a keyword that it does not understand, it cannot safely ignore it, but rather should raise an exception. In the future feature sets will be linked to the Resource Description Framework (RDF) [9].

The simplified feature set representation of a *Please connect me to Bob* goal is shown in the following table.

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
<td>connect</td>
<td>string</td>
</tr>
<tr>
<td>source</td>
<td>Me</td>
<td>person</td>
</tr>
<tr>
<td>destination</td>
<td>Bob</td>
<td>person</td>
</tr>
</tbody>
</table>

Ultimately, goal representation matters for the operations that can be performed on them by *techniques*. Techniques are pieces of code that comply with a certain interface and whose overall purpose is to either solve a goal or split a goal up into several sub-goals. In particular, techniques need to be able to answer the question whether they can be matched against a specified goal. A technique matches a goal if it knows how to satisfy it, i.e. it understands what the goal means. This directly leads to the conclusion that the *matches()* method of the technique interface should do a semantic check on the supplied goal representation. A side effect of that check is data discovery. For example, if a particular technique is supposed to handle the "connect" command, then after trying to match the *Please connect me to Bob* goal, it will know that the first object of the "connect" command is "me" and that the second is "Bob."

The actual approach that a technique might use to match a goal depends on the goal’s representation. For example, if the goal’s representation is in terms of parse trees, then
the technique might try to match the goal against a pre-defined template tree. The template for matching the “connect” goal could be

```
SENTENCE
  |     |
  | please
  |     |
  COMMAND                      OBJECT
      connect              ?source
                                 to
                                 OBJECT
                                       ?destination
```

After the comparison of the goal and the template above, the technique will realize that it in fact can handle this goal. In addition, it will discover that source variable should be bound to “me” and that the destination variable should be bound to “Bob.”

Performing matching for a feature set representation of goals can be done by direct inquiries into the set. The technique for the “connect” command might first want to check whether the “command” keyword is present in the set and then test to see if its value is “connect.” If this passes, then the technique would check “source” and “destination” keywords, and if those exist, then it would bind their corresponding values to the appropriate source and destination variables. If any of the steps fail, the match method will fail indicating that this technique cannot be used for satisfying the specified goal.

The two different goal representations reflect the usual and quite frequent (as the reader will notice later) tradeoff between flexibility and ease of use or simplicity. The parse-tree representation of goals provides with greater flexibility for adding new techniques that are supposed to handle previously unknown goals. Assuming the voice shell used to
process user commands can parse a wide range of natural language sentences, the author of the new techniques just needs to appropriately implement the `matches()` method. For the feature-set representation of goals, the author of the new techniques would also have to implement a plug-in for the voice shell that will be able to correctly convert the uttered command into a feature set before it can be passed as a parameter to the `matches()` method. On the other hand, matching a parse-tree against a number of templates is a more difficult task than picking the appropriate keywords from a feature set.

Goal representations also differ in the placement of the code, which does a conversion from a human-readable description of an object, such as “me”, “Bob”, “the nearest printer,” into a remote object reference understandable by the system. The leaves in the parse-tree representation are the original strings describing the objects, while in the feature-set representation the values are the remote object references. In other words, in the first case, the conversion is done at the technique level, which makes it more flexible but also more difficult for a technique implementer, while in the second case the conversion is done at the voice shell level, which is easier on the technique author, but also limits the lookup capabilities by what was already implemented in the voice shell.

Our assumption was that new types of goals are added to the system much less frequently than new techniques for existing goals; therefore, we opted for the feature-set representation of goals. This also has an additional benefit of unifying procedures for common operations on feature sets, such as serialization and deserialization to and from XML.

For the prototype implementation of the system, we actually didn’t implement the full feature-set representation support. Goals are represented by a single keyword, uniquely distinguishing them between each other, followed by a series of parameters, any of which can be optional: source, destination, and bandwidth.
6.2. Goal Trees

Goal satisfaction process creates a goal tree in order to find the sequence of actions, called a plan, that will achieve the goal with the best utility value termed satisfaction. This tree contains two different alternating (along parent-child link) kinds of nodes. For each given goal there might be several techniques that are capable of handling it; hence, goals and sub-goals correspond to choice nodes. Each technique creates a planlet node, which is called so because it will eventually produce a plan to achieve the goal if that particular technique is selected for this purpose. Instead of satisfying the goal directly, the technique might choose to rewrite it in terms of one or more sub-goals, which are represented as choice nodes in the goal tree. All of this creates the alternating pattern of choice and planlet nodes in the tree, where the root of the tree is a choice node and leafs can be of either type. Leafs that are choice nodes indicate that there were no matching techniques found for the corresponding sub-goals. These sub-goals cannot be satisfied and, thus, have the satisfaction value of zero.

![Diagram of a goal tree]

**Figure 2.** The structure of a goal tree. Nodes in bold is a sample execution sub-tree (only one of the planlet nodes can be selected and all of the choice nodes should be selected).

The execution sub-tree of the goal tree is the sub-tree that corresponds to the best plan for the root goal. This sub-tree is built by selecting only one child of a choice node and all children of a planlet node. The execution is performed by carrying out the actions prescribed by the techniques corresponding to the planlet nodes in the execution sub-tree.
In order to determine for a given choice node which planlet to pick, the satisfaction values are calculated for all the nodes in the goal tree; then the planlet with the maximum satisfaction value is chosen (see Figure 2).

The process of assigning the satisfaction values to the goal tree nodes is similar to the minimax algorithm because it alternates the minimizing and maximizing strategies. The satisfaction value for a choice node is the maximum of satisfaction values for its children, which are planlet nodes. The default behavior for calculating the satisfaction value of a planlet node is to pick the minimum satisfaction of its children, which are choice nodes. This default behavior can be overwritten by the techniques. In fact, it should be overwritten by techniques which correspond to the leaf planlet nodes because there are no children nodes whose satisfaction could be used to calculate the minimum.

6.3. Satisfaction Scale

The main assumption of our algorithm is the existence of a universal scale for satisfaction values. We arbitrarily set it between the values of 0 and 1. The points on this scale do not convey any information in isolation, but rather are meaningful for comparisons only. For example, if one plan for a certain goal has a value of 0.6 and an alternative plan for that goal has the satisfaction of 0.7, this means that the latter is preferred. The value of zero for planlet nodes usually indicates that its corresponding plan could not be executed. The value of zero for choice nodes usually indicates that there is no plan that could be executed to satisfy the corresponding goal (Figure 3).

\[0\]  \[0.6\]  \[0.7\]  \[1\]

lower satisfaction superior alternative higher satisfaction

inferior alternative

Figure 3. Satisfaction scale.

A planlet node's satisfaction is the utility level that the user will achieve if the corresponding plan is executed. A choice node's satisfaction is the maximum utility level
that can be achieved for the corresponding goal. For choice nodes, we are picking the child with the maximum satisfaction because we would like to find the best plan and because we assume that it is possible for each of the planlet nodes to have satisfaction value on the unified scale as described above. The formula for calculating the satisfaction of a planlet node is less obvious; however, no matter what the default behavior is, it can be easily modified by local techniques. The requirement on the formula is that the satisfaction should increase or at least not decrease with the increase in the satisfaction of one or more of the sub-goals. There are many formulas that match this description; for example, minimum, sum, or product functions have the desired characteristics. The default formula of our choice is to find the minimum of satisfaction values of the plan’s sub-goals (Figure 4).
**Watch a movie** (satisfaction = max (0.6, 0.65) = 0.65)
- in a theater (satisfaction = min (0.9, 0.9, 0.7, 0.6, 1.0) = 0.6)
  - pick the movie title (satisfaction = max (0.9) = 0.9)
    - default (satisfaction = 0.9 reflects the task’s tediousness)
  - find theater location (satisfaction = max (0.9) = 0.9)
    - default (satisfaction = 0.9)
  - buy tickets (satisfaction = max (0.7) = 0.7)
    - default (satisfaction = 0.7 reflects the task’s monetary expense)
  - get to the theater (satisfaction = max (0.6) = 0.6)
    - default (satisfaction = 0.6 reflects the unwillingness to travel)
  - watch the movie (satisfaction = max (1.0) = 1.0)
    - default (satisfaction = 1.0)
- at home (satisfaction = min (0.9, 0.65, 1.0, 0.9) = 0.65)
  - pick the movie title (satisfaction = max (0.9) = 0.9)
    - default (satisfaction = 0.9 reflects the task’s tediousness)
  - obtain the movie (satisfaction = max (0.55, 0.65, 0.50, 0.45) = 0.65)
    - buy DVD (satisfaction = 0.55)
    - rent DVD (satisfaction = 0.65)
    - buy VHS (satisfaction = 0.50)
    - rent VHS (satisfaction = 0.45)
      - rent DVD (satisfaction = 0.65) Reflects the cost and the quality of the media and whether the user is able to keep it.
  - insert it into the player (satisfaction = max (1.0) = 1.0)
    - default (satisfaction = 1.0)
  - play the movie (satisfaction = max (0.9) = 0.9)
    - default (satisfaction = 0.9 reflects the drawbacks of a home audio/video system)

Figure 4. A sample goal tree showing the satisfaction calculation algorithm in action. Default technique for some goal means that it is the only technique designed to satisfy that goal. The best plan is in bold and choice nodes are underlined.

### 6.4. Techniques

In order to see how to customize the satisfaction calculation formula, we shall discuss the technique interface and how it fits in the goal satisfaction process. The process starts with the creation of the root goal tree node, which is a choice node with a link to the original goal. The satisfaction operation on the tree involves four consecutive stages: initialize, evaluate, mark, and execute. These stages are represented by four methods in the `GoalTreeNode` class: `initializeNode()`, `evaluateNode()`, `markChosenSubtree()`, and `executeNode()`. The actual implementation of these methods differs based on the kind of the node: `Choice` class for choice nodes and `Planlet` class for planlet nodes. While choice nodes know how to process those
requests independently of the associated goals, planlet nodes rely heavily on the associated techniques to provide the required functionality.

The initialization stage for each node does the required preparation for the later stages. Normally, it would expand that node by creating its children and then initialize some local state variables for later use. To ensure that all of the children get initialized, the parent node asks them to initialize as well at this point. A choice node expands by finding matching techniques for its associated goal and creating a child planlet node for each of them. A planlet node expands by enumerating all of its sub-goals and creating a child choice node for each of them. The initialization behavior for planlet nodes is completely defined by the initialize() method in the technique interface since it is not possible for a generic Planlet class to know what kind of sub-goals a particular technique requires.

The evaluation stage for each node calculates the satisfaction of that node. It first calculates the satisfaction values for all of its children and then finds the minimum to calculate the satisfaction of a planlet node and the maximum to calculate the satisfaction of a choice node. There is an opportunity for techniques to override the default formula by implementing the evaluate() method in their interface. While executing the evaluateNode() method, a planlet node checks to see if the technique calculated its own satisfaction using the evaluate() method, and if so, uses it as the resulting satisfaction of the planlet node.

The mark stage recursively marks the nodes for inclusion in the execution sub-tree. For choice nodes, it marks the node itself and then descends into the child planlet node with the highest satisfaction. For planlet nodes, it also marks the node itself, but descends into all of its children.

What the execution stage does is that it simply carries out the actions comprising the best plan to achieve the original goal. The execution stage follows the execution sub-tree in post-order sequence. In other words, the children are executed first and then their parents. Note that there is no execution code associated with the choice nodes. Planlet nodes get executed by forwarding this request to their corresponding techniques. Techniques
provide code for execution in the \texttt{execute()} method. In order to support techniques which prefer their sub-goals to be executed in a certain order, planlet nodes execute their children in the order they were added.

There are two more methods that are included in the technique interface, which are \texttt{reset()} and \texttt{matches()}. The \texttt{reset()} method returns the state of the world to the state that was before the execution of the best plan. Its presence in the interface is only for the convenience of certain applications, which can undo their actions. For example, in the teleconference application, the goal tree corresponding to the \textit{Please connect me to Bob} goal could be reset by breaking the audio/video connection that was established earlier. In other words, upon receiving \textit{Stop the teleconference} goal, the system could simply locate the old goal tree and reset all of the nodes. Planlet nodes reset themselves by reinitializing their state and also by calling the \texttt{reset()} method on the associated techniques, which gives them an opportunity to break up the constructed connections.

The \texttt{matches()} method answers the question whether a particular technique is capable of solving a certain goal and was described earlier. Currently, it returns a Boolean answer of “yes” or “no” and works in conjunction with the \texttt{getInstance()} method in the technique interface, which returns an instance of the planlet node ready for inclusion in the goal tree. There are actually several ways of implementing the interaction between techniques and planlets.

As was stated before, techniques are ways of solving certain kinds of goals. Planlets, on the other hand, are part of a goal tree and store the state required for solving a particular goal. Hence, the first way of implementing their interaction is to make them two separate Java classes: \texttt{Planlet} class and \texttt{Technique} class. \texttt{Planlet} class implements the methods \texttt{initializeNode()}, \texttt{evaluateNode()}, and \texttt{executeNode()}, which forward these requests to methods \texttt{initialize()}, \texttt{evaluate()}, and \texttt{execute()} in \texttt{Technique} class. In order not to pass any state between a planlet and a technique, there is a separate instance of the technique for each instance of the planlet. The drawbacks of this method are the complexity of maintaining the one-to-one
correspondence between planlets and techniques and the extra space that all of the new instances of techniques require.

It is possible to merge Planlet and Technique classes together, which simplifies the maintenance of the one-to-one correspondence. In this case, the instance of technique is what plays the role of the planlet. However, there is still a need for separate planlet and technique methods. For example, evaluateNode() still must be present since it does some work related to the goal tree before calling out to evaluate(), which the author of the technique overwrites. Another drawback of this merge is a semantic one: the instance of the technique playing the role of a planlet has access to the matches() method, invoking which does not make sense at this stage. What is even worse is that the merge between planlet and techniques implies that techniques can only be used in satisfying goals using the goal-tree approach, which might change in the future. Currently, though, this approach is sufficient for the purposes of the prototype implementation.

Another way of implementing the planlet/technique interaction is to make the matches() method of the technique interface return a specialized version of the Planlet class, which will be able to store the state pertaining to that particular technique. This semantically separates planlets and techniques, but there is still a dependency of techniques on the existence of goal trees for which those specialized planlets are designed. Another problem with this approach is doubling the number of classes for each technique. Now each unique technique class must also have a corresponding unique planlet class.

There is a better approach which simplifies the interaction between techniques and planlets by storing technique code in a static class which can be easily downloaded from a remote machine using an HTTP server and by storing the state needed for a specific goal resolution in a generalized Planlet class. The advantage of this approach is an easier distribution of planlets and techniques over multiple machines and the minimization of the number of Java classes and objects, for there is no need for either the creation of a subclass of Planlet for each technique or the creation of a separate technique instance for each planlet. Another benefit of this approach is that now
techniques are completely separated from planlets, which may render them usable in other planning algorithms, which do not involve goal trees. The drawback of having a generalized planlet class for all of the techniques is that now they cannot store customized technique state. For example, if some specific technique requires storing a variable \( a \) somewhere, it cannot be stored in the technique, since it is a static class in this approach; neither, it can be stored in the planlet directly, since the same planlet class is used for all kinds of techniques. The solution to that is for planlets to add support for feature sets, which can store arbitrary information. Since the concept of feature sets is not quite developed, currently, this approach can only be emulated by storing the local data in a data structure similar to a hash table. However, if hash tables are used, then it becomes quite tedious to store state to and restore state from these hash tables between the invocations of \( \text{initialize}() \), \( \text{evaluate}() \), and \( \text{execute}() \) methods. Additionally, no compile-time data type checks can be performed on data stored in a Java version of hash table. To summarize, this is quite a promising approach given that the feature set infrastructure is fully developed.

Specific technique classes could be implemented either using named classes or anonymous inner classes. Named classes take a bit more space of Java code, but they are much easier to instantiate than anonymous inner classes, since the only support for instantiating anonymous inner classes in Java is through reflection: special complicated syntax for accessing fields and methods of classes known only at run-time. Anonymous classes can be written quite conveniently and compactly, but there are several other limitations placed on them by Java. For example, since they are implicitly inner classes, they cannot have non-final static variables. In addition, in order to be instantiated many times, they must be defined in a static context with a no-arguments constructor [7]. In short, the use of anonymous classes for techniques proves useful only for small techniques with little code in them. Fortunately, our framework does not constrain the choice for picking either named or anonymous classes for techniques: both types can be used in the same application.

In a distributed environment, techniques can be located on one computer (a remote machine) and the tree might be built on another computer (let’s call it local). In such a
case, techniques should be transferred to the machine involved in the goal tree evaluation. One way of transferring them is to instantiate techniques on the remote machine and then transfer them in a serialized form. However, since there is little state that can be initialized remotely, it makes more sense to download only the Java class from the remote machine, and then instantiate it locally. This would result in not worrying about the serialization and also in less network traffic.

Another aspect to note about techniques is that the interface to them is quite opaque, i.e. access to the internal pieces of them can only be done via the invocation of the methods comprising the interface. While it works well for the goal-tree evaluation approach, it might become problematic to use this interface with other kinds of algorithms, which require a direct access to the information about the internals of the technique. One solution would be to use a more transparent data structures rather than to use opaque arbitrary code. However, we again are presented here with a trade-off between flexibility and ease of use. Data structures are easier to manipulate but are less expressive. Using arbitrary pieces of code is extremely powerful, but is not readily amenable to external analysis. David Saff suggests that at least the initialize() method of the technique interface should be replaced with a data structure explicitly listing all of the required sub-goals. This limits the inclusion of custom initialization procedures in the techniques, while at this point it is still unclear how much benefit will be gained by making the initialization stage more transparent.

6.5. Dynamic Goal Tree Reevaluation

Failures are possible at each stage of the satisfaction process, whether it be goal tree initialization, evaluation, or execution stage.

At the initialization stage, errors might occur while creating a certain goal tree node. If a certain node fails to initialize, it throws an expansion exception. The sources for this error can be quite different, one of them being the network problems related to the communication between the distributed components of the system. For a planlet node, the expansion exception means that this node should not be added to the tree. There will be just one alternative less for solving some sub-goal. The exception thrown by a choice
node indicates that a certain sub-goal cannot be solved in principle, which should cause
the exception to propagate further up through the planlet node, which was trying to add
this choice node.

At the evaluation stage, failures might happen not only because of some exceptional
circumstances but also because it might be discovered in the due course of calculations
that a certain sub-goal cannot be satisfied. In such a case, the node which discovered the
failure sets the hasFailed mark on itself with a variable failureReason indicating
failure’s cause. The parent node will inspect this mark on its children while trying to
calculate its own satisfaction. For planlet nodes, if any of their children have failed, it
fails itself. Choice nodes fail themselves only if all of their children have failed. Failure
marks on the goal tree nodes are used when selecting the execution sub-tree since in no
case failed nodes can be executed.

At the execution stage, the exception might happen when trying to execute a certain
action that seemed possible earlier but which is not the case any longer. For example, two
different sub-goals might require the use of a certain resource, which seems available to
both of them at the evaluation stage; however, at the execution stage, only the plan for the
first sub-goal will be able to lock, while the second plan will have to throw an execution
exception. Execution exceptions mark the node which threw it as failing and propagate
further up to the point where an alternative plan can be found, at which point the old
execution sub-tree is reset, the sub-tree reevaluated, and the new execution sub-tree is
executed.

The exceptions happening at the execution stage are tightly related to the dynamic goal
tree reevaluation in response to environment changes. For example, a certain decision
about picking the best plan could have been made based on the information that some
resource is not available, such as hard disk space. However, while the plan is still being
executed, the user might plug in another hard drive into his desktop using the USB port.
The goal tree must be reevaluated in search for a better plan, but the question is how it
will know about this change in the environment.
The current version of our system gets notified from the top. The process that originally built the goal tree reevaluates the whole goal tree after having learned that a certain change in the system might affect the choice for the best execution plan. If a better plan is available, the old plan is reset and the new execution sub-tree is marked and executed. The benefit of this approach is its simplicity but the drawback is that many unrelated goal tree nodes might need to be reevaluated. Ideally, however, the sub-goals not involved in writing to disk should not be reevaluated when a new hard drive is added to the environment.

The proposed better approach is for techniques that make the decisions about the best course of action for solving their corresponding sub-goals to register themselves as listeners to the components whose change in the future should cause the techniques to reconsider their decisions. For example, the technique that made the inquiry into the operating system about the presence of hard drives in the environment might register itself with the operating system as interested in listening to the events that might change the answer to the original query. Of course, the operating system should be able to support listener registration, but it is projected that all of the components in the future will have the hooks for reporting their change to the interested parties. The listener interface is called ChangeListener and contains a single method changeOccurred(). When this method is invoked on the listener, it is similar to throwing an execution exception except that it does not mark the node receiving it as failing. The problem with this approach is that if changes are reported at a rather low level and if there are numerous listeners registered with a certain component, then the functioning of that component might become unacceptably slow. Also, a certain single change in the environment might trigger many other changes, which could cause the tree to be reevaluated many times, while it should be done only once. Solutions to both of these problems require further research.

To summarize, the current version of the interfaces for GoalTreeNode and Technique classes are shown in Figure 5 and Figure 6.
6.6. Contexts

When the system is trying to find a technique for some goal, it needs to have the means of browsing through a catalog of techniques and picking a matching one. The service for the Oxygen platform that can be used to lookup techniques that were previously

```java
public abstract class GoalTreeNode implements Serializable {

    // goal tree interface
    abstract void initializeNode () throws ExpansionException;
    abstract double evaluateNode ();
    abstract void markChosenSubtree ();
    public boolean isChosen ();
    abstract void executeNode () throws ExecutionException;
    void resetNode ();

    // failure handling
    void fail (String reason);
    public boolean hasFailed ();
    public String getFailureReason ();

    // memoized value of satisfaction
    double getSatisfaction ();
    void setSatisfaction (double sat);

    // tree interface for enumerating children and access the parent
    public int size (); // number of children
    public GoalTreeNode child (int i);
    void add (GoalTreeNode n); // add another child
    public GoalTreeNode getParent ();

    // access the associated goal or sub-goal
    abstract Goal getGoal();
}
```

Figure 5. GoalTreeNode interface.

```java
public abstract class Technique extends GoalTreeNode {
    public boolean matches (Goal goal);
    public Technique getInstance (Choice parent);
    void initialize () throws ExpansionException {};
    void evaluate () throws RemoteException {};
    void execute () throws ExecutionException, RemoteException {};
    void reset () throws RemoteException {};
}
```

Figure 6. Technique interface.

6.6. Contexts

When the system is trying to find a technique for some goal, it needs to have the means of browsing through a catalog of techniques and picking a matching one. The service for the Oxygen platform that can be used to lookup techniques that were previously
registered with it by technique authors is called the *O2S Registry*. O2S Registry can also be used to lookup *pebbles* (see further) and *O2S hosts*. (The word “O2S” can be omitted in the further discussion but is still implied.)

However, the number of techniques can be tremendous, so it is impossible to expect that all of them will be registered with the same registry. Therefore, there needs to be a system for organizing registries in some sort of hierarchical structure. The basic unit of this structure is called a *context* because it contains some local information for a certain domain. Contexts store information about the registries that they trust and also have ways of finding out about other surrounding contexts. For example, user Ben’s personal context might specify that he is a member of the organizational context for MIT LCS and also a member of a Bitdiddle family context.

Contexts can be asked to satisfy a goal and manage the resulting goal tree (e.g. delete or update it), form themselves into hierarchies, and forward lookup requests to registries based on those hierarchies.

The interface for goal satisfaction and the management of the related goal trees includes at least the following methods: `satisfy(Goal)` and `delete(Goal)`, where goals are represented uniquely using one of the representations discussed in section 6.1. `satisfy(Goal)` method builds the goal tree, evaluates it, and then executes the best plan for it, which corresponds to the execution sub-tree. The context remembers about the goal tree as long as the `delete(Goal)` method have not been called. Until then the presence of the goal tree in the system indicates that user’s original intention is still valid and that the context should strive to maintain the execution of that goal by dynamically re-evaluating the goal tree and executing the best possible plan at any given moment. Even if at some point all of the plans fail, the context should not abandon the goal until the user says so. This is helpful for persistent goals, such as a long-running teleconference goal, which should survive despite transient network failures. Of course, the requirement for an explicit call to the `delete(Goal)` method does not exclude making this call part of a technique for solving a short-lived goal. For example, the technique for solving *Play music file X* goal might consist of several sub-goals:
1) locate file X,
2) locate a player for files of type (X),
3) play the file in the player,
4) stop the player,
5) delete the goal tree.

In this example, if the sub-goals 1 through 4 executed successfully, then goal 5 will be executed, which will destroy the goal tree and the context will no longer try to play file X.

Contexts should be able to form hierarchies by maintaining the parent-child links. To this end, they should support methods addChild (Context), deleteChild (Context), addParent (Context), and deleteParent (Context), which are self-descriptive. Note that it is possible to build a hierarchy with these methods, which does not necessarily have to be a tree, i.e. each context can have more than one parent. This is useful in situations where a certain context should really be part of more than one larger context as was the case with Ben’s context above.

### 6.6.1. Context Lookup Algorithm

Contexts should forward the lookup requests to the registries in the order that follows the context hierarchy. In other words, when the context receives one of the findTechniques (FeatureSet), findPebbles (FeatureSet), or findHosts (FeatureSet) requests, it returns an iterator, which iterates over a set of possible objects matching the feature set in the order prescribed by the context hierarchy and local policies for registry lookups. Those policies can be quite flexible. First of all, they can differ based on the type of the requested objects. Secondly, they can prescribe whether the parent’s or the child’s registries (and the objects registered with them) take precedence. Thirdly, the policies indicate which of the child’s parents takes precedence. Lastly, it should be possible to specify which of the local registries is primary.

In Figure 7, context A has context B as parent, B has three parents C, D, and E, and C also has an additional child context F. All of these contexts have an associated registry,
while D, in fact, has three associated registries. Suppose that for a certain lookup query all of the objects a, b, c, d1, d2, d3, and e matched (F’s registry did not have any matching objects), then A can return these objects in any of the following orders determined by local lookup policies:

- a, b, c, d1, d2, d3, e
- a, b, c, d3, d2, d1, e
- a, b, e, d1, d2, d3, c
- a, b, e, d3, d2, d1, c
- c, d1, d2, d3, e, b, a
- c, d3, d2, d1, e, b, a
- e, d1, d2, d3, c, b, a
- e, d3, d2, d1, c, b, a
public interface Context extends Remote {

    // for goal satisfaction
    FeatureSet satisfy (Goal g) throws RemoteException, ExpansionException;
    void delete (Goal g) throws RemoteException;

    // for context hierarchy maintenance
    void addChild (Context c) throws RemoteException;
    void addParent (Context c) throws RemoteException;
    boolean deleteChild (Context c) throws RemoteException;
    boolean deleteParent (Context c) throws RemoteException;

    // for looking up objects in the context hierarchy
    Iterator findTechniques (FeatureSet fs) throws RemoteException;
    Iterator findPebbles (FeatureSet fs) throws RemoteException;
    Iterator findHosts (FeatureSet fs) throws RemoteException;
}

Figure 7. Possible object lookup within context hierarchy.

Figure 8. Context interface.

To summarize the previous discussion, proposed context interface is shown in Figure 8.
7. Pebbles Layer

The sequence of actions that comprise the best plan for certain goals is a description of a connection diagram. For the teleconference application, for example, the result of the planning phase is a directed graph, which describes the locations of the distributed components that process audio and video streams and their interconnections. These components are called pebbles.

Pebbles are pieces of code designed to carry out a certain task described by a specification. Specifications strictly define how the pebbles conforming to those specifications should behave. They not only unambiguously specify the interface but also provide a human-readable description as well as a number of test cases that any conforming pebble should pass. There can be several alternative pebble implementations of the same specification by different authors. Because of the specification, they can all be substituted for one another without breaking the semantics of the execution plan, while allowing the planning layer, which assembles the pebbles, to pick the one with the desired characteristics, such as optimum code size, hard drive space, computation time, or network bandwidth, especially since there is usually a trade-off between those. Examples of pebbles include pebbles, which record or play audio, multiply or multiplex data streams, or convert video to other formats.

The basic view of a pebble is that it is a portable component (i.e. can be located and run on different network machines), which might have a number of inputs and outputs. Pebbles, which do not have any inputs are called sources, those that have no outputs are called sinks, and those that have both can be either filters and converters (one input to one output), multipliers (one input to many outputs), multiplexers (many inputs to one output), or some combination of the above.

Pebbles are usually instantiated on specific machines. Machines that are designed for the Oxygen platform and which support pebbles are called O2S hosts. They are registered with the O2S registries and are looked up by contexts as was described in section 6.6.1. Hosts support starting and stopping pebbles. The entity, which wants to install a pebble on a certain host (usually a technique at the execution stage), sends a “start pebble”
message to that host along with the pebble’s specification (or the URL where it can be found) and a reference to itself, so that the run-time changes and errors could be reported back to the installing entity.

The return value from the `installPebble()` call is a proxy to the actual pebble running on the host. It supports several operations including querying the status of the pebble installation, connecting it to another pebble, and stopping the pebble’s execution. The returned status can be either “completed,” indicating that the pebble’s installation was successful and that it was run to completion successfully, or “running,” meaning that the pebble was installed successfully and is still running, or “error” with a reason for either installation error or pebble execution error. “completed” status is used for synchronous calls, where a pebble is required to accomplish only a small short-lived task and soon return back to the caller. “running” status, on the other hand, is used for asynchronous calls, where pebbles are installed for a long time and must be executed in parallel with other calls. This asynchronous model is the basis for most pebble installations and, therefore, allows further interaction with pebble proxies besides querying the status. For example, pebble proxies can be asked to connect their corresponding pebble to another pebble. Calling `PebbleProxy.connectTo(PebbleProxy)` will result in forwarding this request to the actual pebbles and will connect them in such a way that the output of the first pebble is directed towards the input of the second pebble. This corresponds to a directed edge in the connection graph. After it is determined by the system that the execution of a certain pebble is no longer desired, its proxy receives the request to stop the pebble.

When someone asks a host to install a pebble, which already exists and is running on the host, the host might simply return the proxy to the old pebble. This is useful, for example, for pebbles which support multiple inputs or outputs. It is also useful for locking resources. Adding extra connections to the pebbles, which do not support multiple inputs or outputs, will result in an execution exception, which will cause a certain part of the goal tree to be reevaluated and re-executed.
Pebble layer and the whole distributed component assembly concept is currently being actively developed by Prof. Steve Ward and Umar Saif. The interface for hosts, pebbles, and pebble proxies is shown in Figure 9.

```java
public interface Host extends Remote {
    PebbleProxy installPebble (PebbleSpecification ps) throws RemoteException;
}

public interface PebbleProxy extends Remote {
    final int COMPLETED = 1;
    final int RUNNING = 2;
    final int ERROR = 3;

    // status query
    int getStatus ();
    String getErrorMessage();

    // connecting pebbles
    void connectTo (PebbleProxy pp) throws RemoteException;

    // stop the corresponding pebble
    void stopPebble () throws RemoteException;
}

public interface Pebble {
    // connecting pebbles
    void addInput (PebbleProxy pp) throws ExecutionException;
    void addOutput (PebbleProxy pp) throws ExecutionException;

    // install and run the pebble on the host, returns status
    int run (Host h);
    String getErrorMessage();
}
```

Figure 9. Host, Pebble, and PebbleProxy interfaces.

8. Teleconferencing Application

The teleconference application (the first version was implemented in Curl by Prof. Steve Ward) represents a proof of a concept that the local decisions in the goal tree can be factored out of the goal-oriented planning system in the form of highly customizable techniques.

This application allows one user to establish a teleconference connection to another user. This process usually starts with the first user (let’s call her Alice) speaking “Please connect me to Bob” into the voice shell on the host that is nearest to her, handheld, for
example. The voice shell recognizes the speech and parses the input converting it to one of the goals representations described in section 6.1. This goal is then submitted to the local context, the context to which the host with the voice shell belongs to. The context builds a goal tree and goes through the stages of evaluating and executing it. The result of the last stage is the connection graph or diagram, which shows how the four audio/video channels are established: audio and video channels from Alice to Bob as well as audio and video channels in the reverse direction. Pebbles are installed and connected according to this diagram. If there are any changes in the environment, which might render the connection diagram no longer valid, the goal tree is updated, reevaluated, the old plan is reset and the new one is executed. This process repeats until the goal is removed from the system by removing the corresponding goal tree.

We built a simple demo, which demonstrates the abilities of the teleconference application and of the goal-driven planning in general. This demo is GUI-based and on the main screen it lists the buttons for stepping through a particular scenario. It also shows the goal tree with the nodes showing their names, satisfaction, and whether they belong to the execution path (green color) or whether they failed (red color). To distinguish between the choice nodes and the planlet nodes, the goals are enclosed in figure brackets. Additionally, all planlet nodes have an “OR” icon near them indicating that either of them can be chosen for satisfying the parent goal, and all the choice nodes have “AND” icon besides them indicating that all of them must be satisfied before the parent technique can execute (see Figure 10).
Figure 10. Teleconference demo scenario, step 1.

The scenario develops as follows. When Alice first asks the system to establish the teleconference with Bob, she only has a handheld device (H21 in Oxygen terminology) with her without a camera and a cell phone. This goal is received by her context and is entered as {Teleconference Alice, Bob}. After looking up the matching techniques, which can satisfy this goal, Alice’s context comes up with two named “Audio Only” and “Full Audio/Video.” As part of the initialization stage, these techniques specify their sub-goals and ask the context to find matching techniques for those as well. “Audio Only” techniques says that it only requires the satisfaction of {ConnectAudio Alice, Bob} sub-goal, while “Full Audio/Video” techniques requires the satisfaction of both {ConnectAudio Alice, Bob} and {ConnectVideo Alice, Bob} sub-goals. The only matching technique for {ConnectAudio Alice, Bob} goal is “Internet Audio”, which specifies the requirement for a duplex audio channel by listing sub-goals {AudioLink Alice, Bob} and {AudioLink Bob, Alice}. Similarly, the technique for {ConnectVideo
Alice, Bob} goal called “Simple unidirectional links” requires the sub-goals {VideoLink Alice, Bob} and {VideoLink Bob, Alice}.

{AudioLink X, Y} goal can be satisfied by only one technique: “100MBit LAN uncompressed,” which needs {getAudioSrc X}, {getAudioDest Y}, and {getConnection X, Y} sub-goals. The technique for the first sub-goal finds an appropriate pebble that can record audio input for X and installs it on some host near X. The technique for the second sub-goal does a similar action but this time for playing the audio for Y on some host near Y. The technique for the third sub-goal connects the two pebbles together using a network connection installing an intermediate pebble between the two to convert the data stream if the output format of the audio source pebble is different from the input format of the audio destination pebble.

{VideoLink X, Y} sub-goal can be solved by either of the three techniques: “No Link,” which is there to allow the sub-goal to be left unsolved at the cost of zero satisfaction value, “Still Picture,” which looks up a still picture for X in an archive and sends it to the video destination pebble for Y, and “LAN compressed” techniques, which is quite similar to the “100MBit LAN uncompressed” technique for {AudioLink X, Y} goal. The difference is that “LAN compressed” technique for {VideoLink X, Y} goal discovers and installs video analogs of the source and destination pebbles and also requires that the bandwidth of the connection between X and Y should exceed 500Kbit/s.

If the initialization stage completes top-down, then the evaluation stage is performed bottom-up. First, the low-level techniques for goals that require source and destination pebbles and connections between them are evaluated. They normally set their satisfaction to 1.0 overriding the default formula since it cannot be applied to the nodes without leafs. Then, their parent nodes calculate their satisfaction. “100MBit LAN uncompressed” and “LAN compressed” techniques multiply the result of the default formula (the minimum of satisfaction values of children) by 0.99 to take into account the effects of network imperfection such as dropped packets or network congestion. Note that “LAN Compressed” technique actually failed for {VideoLink Alice, Bob} goal because Alice does not have a camera and, thus, the installation and execution of a video source pebble
is not possible for Alice. “Still Picture” technique calculates the satisfaction for 
{VideoLink X, Y} goal similarly to the “LAN compressed” technique except that at the end it divides the resulting value by three indicating that it is only one third as satisfactory as a live video link. Techniques for solving a duplex connection goal set the satisfaction of their corresponding planlet nodes to the average of satisfaction values of the two half-duplex connections. This is the reason for {ConnectVideo Alice, Bob} choice node to have the satisfaction value of 0.66. All the other nodes in the tree calculate their satisfaction values according to the default formulas. The {Teleconference Alice, Bob} choice node compares two children planlets and picks the one with the highest satisfaction, i.e. “Full Audio/Video.”

Similarly to the initialization stage, the marking stage descends into the tree top-down. It starts with the root choice node and descends into that planlet node, which has the highest satisfaction, simultaneously marking it as part of the execution sub-tree. The marking algorithm also descends into all of the choice nodes, whose parent planlet node was included in the execution sub-tree, and adds them to the execution sub-tree. In our scenario, “Full Audio/Video” technique is marked for satisfying the {Teleconference Alice, Bob} goal, “Still Picture” technique is used for satisfying the {VideoLink Alice, Bob} goal, and “LAN Compressed” technique is selected for satisfying the {VideoLink Bob, Alice} goal.

The execution stage again proceeds traversing the tree bottom-up. In this particular case, though, the order of execution for techniques does not matter because the result is a static connection diagram depicted in Figure 11 (it appears by clicking on the “Diagram” button on the main screen). It is static in the sense that whether the audio connection or the video connection is established first does not change the outcome. The first line in the diagram shows the path for the audio stream from Bob to Alice. Each node in this path consists of two parts: the name or the picture of the pebble corresponding to that node and also the name of the host on which it is running. Bob’s voice is recorded by a microphone pebble running on the desktop in Bob’s office. It is then sent to the LAN-send pebble, which prepares the audio stream for sending it over the network. The Bridge pebble transforms the network stream from the LAN-send pebble into a format accessible
by an AIR-rcv pebble running on Alice’s handheld. The decoded audio stream from the
AIR-rcv pebble is then played on the handheld’s speaker. The connection path for the
audio stream from Alice to Bob is structured analogously. The video stream path from
Bob to Alice is similar to the paths of audio streams except that video codecs (coder-
decoders) are used on both ends to make the size of the video data as small as possible.
Since the Alice does not have a camera, the video connection from her to Bob is not
possible. The corresponding video path shows that a still picture is pulled from a picture
archive on Alice’s handheld and it is then sent to the screen on Bob’s office desktop
computer.

![Connection Diagram]

**Figure 11.** The connection diagram for the teleconference demo scenario, step 1.

To help visualize the goal tree, it is possible to see its 3D version by clicking on the “3D”
button. What appears is 3D representation of the tree that can be rotated, translated, and
zoomed in and out (Figure 12). It is much easier to see the relationship between the goal
tree nodes, when they are shown in 3D. Since the names of the nodes can unnecessarily
obstruct the view, they are not shown in the 3D view; however, by clicking on them it is
possible to highlight corresponding lines in the 2D view and also see their names in the
status bar. Nodes are colored in three colors: nodes in the execution sub-tree are shown
green, failed nodes are red, and all the other nodes are blue. To distinguish between the
choice nodes and the planlet nodes, the former are represented by spheres and the latter
are shown as cubes.
This 3D view of the goal tree is built in two steps. First, a cube of 3D space that will nicely fit in a window is split into as many equal layers as is the depth of the goal tree. Each level of the tree will occupy one layer of the 3D cube. Second, the locations of the nodes of the goal tree in 3D are set according to their level in the tree and also on whether their type is a choice node or a planlet node. The root node is situated in the center of the top layer of the 3D cube. The remainder of the cube is then split equally into as many columns as there are children planlet nodes in the root choice node. If seen from the top, the column cuts are made vertically. In other words, the remainder of the 3D cube is distributed equally between the children from left to right. A child planlet node takes its column, puts itself in the center of the column’s top layer, and then splits the rest of the column equally into as many sub-columns as child choice nodes it has. This time, however, the split is done horizontally if seen from the top or from back to front if seen from the front. In other words, the choice nodes are distributed in the perpendicular direction to the planlet nodes. The process repeats until the leaf node is situated in the right place in the 3D space.
In the second step of the scenario, Alice walks into a well-equipped conference room with full audio and video capabilities. The updated goal tree and the new connection diagram are shown in Figure 13 and Figure 14. The goal tree shows that now the “LAN Compressed” techniques succeeds with a value of 0.99 for the \{VideoLink Alice, Bob\} goal. The satisfaction of the root \{Teleconference Alice, Bob\} goal is increased because of that to 0.99, which is compatible with the fact that the quality of the teleconference connection improved compared to the quality in the first step of the scenario. The updated connection diagram also shows that all of the audio/video inputs and outputs switched from Alice’s handheld to their analogs in the conference because presumably they have better quality.
In the third step of the demo, Bob decides to watch his favorite TV show on his only display, which precludes any Alice's video data from reaching him. Thus, the \{getVideoDest Bob\} goals fail causing the "Still Picture" and "LAN Compressed" plans to fail as well. Only a half-duplex video connection is now possible resulting in the
satisfaction value of 0.5 to propagate up to the root goal (Figure 15). The connection diagram in Figure 16 shows that the video path from Bob to Alice is eliminated altogether.

![Connection Diagram](image)

**Figure 15.** Teleconference demo scenario, step 3.

In the fourth step, Alice decides to drive home where the local area network is no longer available. Now, her only means of communication is the cell phone. The planning system immediately notices that and reestablishes the communication paths as shown in Figure

![Connection Diagram](image)

**Figure 16.** The connection diagram for the teleconference demo scenario, step 3.
Figure 17. The 2D view of the goal tree (Figure 18) shows that the satisfaction of the \{VideoLink Bob, Alice\} goal is no longer possible resulting in the satisfaction of the “Full Audio/Video” plan to be zero. \{Teleconference Alice, Bob\} choice node, therefore, selects “Audio only” as the plan marked for execution, which was available previously as well but was never selected because of its low satisfaction value. This time, however, there are no better alternatives, and the teleconference is continued with a 0.10 satisfaction value.

![Connection Diagram](image)

**Figure 17.** The connection diagram for the teleconference demo scenario, step 4.

This teleconference demo scenario is useful for showing the interplay of the goals and techniques for dynamically updating the best plan for satisfying the teleconference goal. The greatest benefit of this system from the user standpoint is that he or she never needs to tell the system to turn on or off any devices or to reroute the audio or video streams. In fact, the user does not even need to know how the audio/video streams are routed at all. Once the goal is entered into the system, it tries to maintain that goal surviving through hardware and software changes or changes in the users’ locations. For reference, the full source code of the techniques that are driving the teleconference demo can be found in the Appendix.
8.1. Use for Debugging

The 2D and 3D goal tree views as well as the connection diagram GUI tool are useful not only for the teleconference demonstration but also as a debugging tool. Even though we added a unit test framework into the system, i.e. the framework in which particular methods or classes can be tested separately, it proved useful to use the GUI tools as well because bugs introduced into the code frequently show up as incorrect satisfaction values in the goal tree view or as unreasonable communication paths in the connection diagram. The system provides for an easy way of tracking down the sources of errors. When a node fails, it stores a textual message describing the reason for the failure. These failures propagate up in the tree, and, thus, it is possible to clearly see, be able to explain, and, if needed, to drill down to the sources of problems. The GUI tools visualize failure causes as tips that appear near the mouse pointer if lingered over failed nodes (see “Alice isn’t accepting incoming video” tip in Figure 18).
In order to use visualization tools separately from the rest of the planning system, they were moved to a separate GUI package and implemented as Java applets, which can inspect the state of some goal tree being evaluated by a remote context. The download of remote goal trees is done through Java RMI and serialization, which is described next.

8.2. Distribution through Java RMI

Java RMI stands for Remote Method Invocation and is a Java incarnation of remote procedure calls (RPCs). As any other RPC system, RMI supports synchronous calls to be made from an object located on one machine to an object located on another machine in a manner that is quite similar to making local calls between the objects located on the same machine. This functionality is achieved through an extra layer of abstraction, which is responsible for network communication. It is possible to build this functionality into both the caller, so that it knows how to make procedure calls over a network, and into the callee, so that it knows how to receive the calls coming from the network, but Java RMI, on the other hand, requires little modification to the local objects making the call and instead splits the remote object into several parts.

Suppose, for example, that two objects Alice and Bob used to communicate with each other locally using a native Java interface for local calls. Next, we decide to put Bob on another machine, but the interaction between Alice and Bob should stay as similar to the local communication as possible. In particular, we don’t want either Alice or Bob worry about marshalling or unmarshalling the arguments to and from network streams. In fact, we want them to know as little as possible about the existence of the network between them. It should be hidden in an extra layer, and as was mentioned above, RMI goes along the path of modifying Bob object for this purpose. Bob is renamed into BobImpl and now extends RemoteObject class, which hides the network layer on one of the endpoints of the network call. RMI compiler then automatically creates the class for the other endpoint of the network call and calls it BobImplStub, which is the object that Alice communicates with and which marshalls the arguments to BobImpl, waits for a reply, unmarshalls the return value, and returns it to Alice. Of course, Alice does not want to know whether she is dealing with BobImplStub or
whether the communication is happening locally and directly with BobImpl. Both of these classes, therefore, implement the same interface called Bob (which explains the name for BobImpl as an implementation of Bob interface and BobImplStub as a proxy to BobImpl running on a remote host). Bob interface must extend Remote interface and all of its methods must declare to throw a RemoteException, which might happen because of network problems. The only change that needs to be done to Alice is to add the ability to handle remote exceptions and also to add some bootstrapping procedure for obtaining the reference to the remote object Bob.

Locating the remote object can be done in two ways: it can be either received from another object or looked up in an RMI registry. For example, if we want to establish the link between Alice and Bob, Bob could be registered with the RMI registry on its machine, and Alice just needs to know Bob’s machine address and the name under which Bob registered to lookup it up. For our planning system, only contexts and O2S registries need to register with RMI registries. Remote references to all the other entities in our system (such as O2S hosts or pebbles) can be obtained from them.

Java RMI went the way of requiring local objects, which would like to interact with remote objects, to explicitly handle network exceptions that might occur during this interaction. In Java jargon, RemoteException is a checked exception. The idea behind this decision was that network problems are unavoidable and once in a while the caller might experience them. If it is not prepared to handle these exceptional circumstances, the semantics of the code could be broken. Of course, it could be left up to the run-time system to deal with remote exceptions; however, no one knows better than the caller what to do if a certain network error occurs. In some cases, it might be better to ignore the exception, in other cases, the remote call should be attempted again, yet in other cases, the program should simply abort with a message to the end user.

The use of explicit remote exceptions, though, presents a problem for a highly distributed system such as ours. Many callers are not prepared to handle remote exceptions and simply re-throw them to the callers that invoked them. As a result, almost all of the methods in such a system declare that they can throw a RemoteException, which
makes it hard to read the code and also to understand where these exceptions will finally be caught and properly handled.

Another problem with the RMI concerns the `toString()` method, which is used in Java for all objects to convert them to a string representation. This method is useful for debugging, logging, and visualization. In our system, for example, the descriptions of the nodes that show up in the 2D view of the goal tree are obtained using the `toString()` method. The conflict between the `toString()` method and Java RMI is that `toString()` method is declared not to throw any exceptions but all the methods in remote interfaces are forced to declare to throw a `RemoteException`. As a result, `toString()` method cannot be included in any remote interface; instead, alternative methods with the same function must be included, for example, `getString()`. It is inconvenient, though, for pieces of code that deal with both local and remote objects and need to obtain their string representations since they do not know which method to call. The solution that we used for this problem was to write a utility method which switches between `toString()` and `getString()` methods based on the type of the object.

### 8.2.1. Serialized vs. Remote Objects

Objects that can be passed as arguments to remote methods are either passed by reference or by value. Remote objects (those that can accept remote calls) are passed by reference while those objects whose state can be serialized, i.e. represented in a form suitable for transporting over the network, are passed by value; attempts to use any other kinds of objects in a remote call will result in a run-time exception. The decision as to whether pass objects by reference or by value is determined by several factors. The problems associated with both approaches are summarized in Table 1. Mainly, the decision whether to make an object remote or serializable rests on the type of entity that it represents. If the object represents some remote entity, such as an O2S host or a running pebble, it should be remote. On the other hand, if the object represents an entity that is not bound to any particular computer, it should be serializable instead of remote.
Problems with Remote Objects

1) more code to write to implement a remote object (the interface and the implementation need to be written and the implementation stub needs to be compiled)
2) slower interaction with a remote object since it requires network communication and marshalling and unmarshalling of arguments
3) remote exceptions that permeate all code and need to be dealt with

Problems with Serializable Objects

1) no access to the resources of the computer where the object was originally created
2) serialization of objects with a lot of internal state can be quite slow and might take significant network resources
3) duplicates of serialized objects are hard to detect because they look as different instances of the same Java class after each deserialization

<table>
<thead>
<tr>
<th>Problems with Remote Objects</th>
<th>Problems with Serializable Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) more code to write to implement a remote object (the interface and the implementation need to be written and the implementation stub needs to be compiled)</td>
<td>1) no access to the resources of the computer where the object was originally created</td>
</tr>
<tr>
<td>2) slower interaction with a remote object since it requires network communication and marshalling and unmarshalling of arguments</td>
<td>2) serialization of objects with a lot of internal state can be quite slow and might take significant network resources</td>
</tr>
<tr>
<td>3) remote exceptions that permeate all code and need to be dealt with</td>
<td>3) duplicates of serialized objects are hard to detect because they look as different instances of the same Java class after each deserialization</td>
</tr>
</tbody>
</table>

Table 1. Drawbacks of passing arguments by value or by reference in a distributed system.

Because of the reasons mentioned above, the only objects that were implemented as remote are O2S hosts, O2S registries, contexts, and pebbles. The other objects that need to be passed as arguments in remote calls have serialization support for them; specifically, features sets, goals, goal tree nodes, and goal trees are serializable.

A hard choice was made for goal trees. On the one hand, they contain lots of data that will take a long time to be serialized over the network. On the other hand, this might not be done too frequently, and, after the local deserialization, the goal trees might be used extensively for GUI visualizations in 2D and 3D views. Had we made goal trees be remote objects, the graphical interaction with their visual representations would have been unacceptably slow to users. There is, however, another problem with serializable goal trees. Goal tree views (both 2D and 3D) should be able to match their own representations of the goal tree between each other and between them and the original goal tree representation received as a result of the deserialization from a remote call. This is needed when, for example, the user asks for more information about a certain goal tree node or when the selection of that node in one of the views should be highlighted in the other view. This matching can be done by 2D or 3D goal tree nodes maintaining a link to the original goal tree node. However, when an updated deserialized version of the goal tree is received by the GUI tools, this matching must be remapped, which is error-prone, especially when the structure of the tree changes or even when the order of the nodes changes.
9. Suggestions for Further Work

9.1. Local Pebble Interaction Improvement

There are a number of directions for possible optimizations of the current system. One direction is to improve the interaction of pebbles located on the same O2S host. Because of a common interface for both remote and local objects, Java RMI supports working with all kinds of objects in the same manner, and no further work needs to be done to eliminate network communication between two objects located on the same machine, provided that they are instantiated within the same Java virtual machine (JVM). Using objects Alice and Bob from our previous examples, it does not matter for Alice object whether it is working with BobImpl object or with BobImpl_Stub object since both of them implement the same interface Bob.

9.2. Goal Tree Size Minimization

9.2.1. Using “Smarter” Techniques

Another optimization concerns the structure of the goal tree. As the reader could have noted from our teleconference application demo, certain parts of the tree are duplicated in several places. For example, \{ConnectAudio Alice, Bob\} branch is repeated twice: the first time under the “Audio Only” plan and the second time under the “Full Audio/Video” plan. One solution to this is to make a combined technique for the \{Teleconference Alice, Bob\} goal, which requires both \{ConnectAudio Alice, Bob\} and \{ConnectVideo Alice, Bob\} sub-goals, but add two new techniques for solving those goals that will permit them to be left unsatisfied. These techniques can be called the “No audio” technique for solving the \{ConnectAudio Alice, Bob\} goal and the “No video” technique for solving the \{ConnectVideo Alice, Bob\} goal, both with the satisfaction value of zero. This approach was actually used to minimize the size of the tree for \{VideoLink X, Y\} goal by providing a technique called “No link.”

9.2.2. Merging Duplicates

Another way of minimizing the size of the tree is to merge duplicate nodes together, but the resulting data structure can no longer be called a tree since its nodes might have more
than one parent. This structure should properly be called an acyclic graph. Dealing with an acyclic graph can be a bit more difficult in terms of data propagation in the bottom-up direction and also in terms of visualizing it in 2D and 3D views, but this optimization direction seems inevitable since it is not reasonable to expect of technique authors to be smart enough to eliminate all kinds of duplicates in the tree.

### 9.2.3. Caching by Contexts

Goal tree optimizations can also be performed via caching at the context level. When a context receives a request to satisfy some goal, it builds a goal tree and goes through the other steps of solving it, but does not discard the goal tree, caching it for future use instead. This way when techniques expand the sub-trees for their sub-goals, they can check with the local context to see whether it knows about a solution to that sub-goal. If it does that sub-goal is not expanded but is rather replaced with a reference to the goal tree, which already exists in the context’s cache. Caching old goal trees not only reduces the size of the future trees but also speeds up the calculations since the old best plan might turn out to be quite close to the new one.

### 9.2.4. Parameterization

A sudden increase in the size of the goal tree might be due to the existence of many quite similar ways of doing something. For example, the goal \{Eat\} might have many matching techniques of the form “Eat at a restaurant A”, “Eat at a restaurant B”, etc. Each of these techniques might contain the same three sub-goals: \{Get to the restaurant X\}, \{Order and eat the meal\}, and \{Pay for the meal\}. The difference between these sub-goals across all of the “Eat a restaurant X” techniques is only the name of the restaurant. To significantly reduce the size of this goal tree and also to free technique authors from writing a separate technique for each new restaurant one could write a single technique “Eat at a restaurant,” which will parameterize all of the restaurants using one variable, let’s call it X. In the evaluation stage, “Eat at a restaurant” technique will iterate over all possible values for X and descend into its three child sub-goals passing X as a parameter as many times as there are restaurants in a database of restaurants. This is another opportunity for feature sets to prove them useful. If a technique’s state is stored in the
form of a feature set and is passed as an argument to initialize(), evaluate(), and execute() methods, then adding one more “feature” describing the name of the restaurant to that feature set is easy.

9.3. Dynamic Libraries of Techniques

Further work is required in providing support for the dynamic addition of new techniques into the system. Currently, when the goal tree is updated, only the satisfaction values are reevaluated. However, if new techniques are added into the system that might solve current goals, new branches in the goal tree should appear. Analogously, those branches of the tree should be deleted, the corresponding techniques of which have been revoked from the system. Going through the full context lookup algorithm described in section 6.6.1 for each sub-goal upon the reception of a notification that the technique set has changed for some context is not practical. Instead, a system similar to the dynamic goal tree reevaluation described in section 6.5 should be used in this case, i.e. when a new technique is added to some context, this context should notify all those parties that have asked for similar techniques in the past about a possible update.

9.4. Improvements in Satisfaction Calculations

It also appears necessary to be able to adjust the formulas for satisfaction calculation over time without rewriting the code for techniques. This adjustment could be made automatically whenever the planning system detects that the user prefers a certain plan better than the one with the highest satisfaction value. Preferably, these adjustments should be saved as preferences in the user’s context so that the goal trees built for the future goals can take those preferences into account.

The main assumption behind the satisfaction calculations is that the calculation of the satisfaction value for some node can be completely determined by knowing the satisfaction values of its children. In the goal tree this means that the best plan for a certain goal is a combination of the best plans for it sub-goals. However, this might not always hold true, as the best plan for a parent goal might be a combination of sub-optimal plans for the sub-goals. A possible direction towards solving this problem is to use feature sets as the return values from the evaluate() method. This feature set for our
simple teleconference application demo will contain only the satisfaction value, but for other applications it might contain a number of satisfaction values together with pointers to their corresponding plan. Analyzing this information from several sub-goals, a technique might be able to pick the best combination of the sub-optimal plans that will yield the best plan for the parent goal.

9.5. Security

Security considerations have not been addressed at all in the current implementation of the planning system while they should be taken into account in a system striving to achieve any real-world use. The main points for security attacks or privacy compromise are located in O2S registries and O2S hosts. Registering malicious techniques or asking hosts to install insecure pebbles could lead to the destabilization of the whole system.

10. Conclusion

In the course of our research, a goal-driven planning system was designed that meets the requirements of Project Oxygen for its pervasiveness and updatability. The major advantages of this system are its generality, extensibility, and applicability for a wide range of applications. The extensibility is provided by the notion of techniques, which represent local decisions about the planning process. Techniques can be written by independent authors and added into the system without requiring its restart. The drawback of our planning system is that it might be slower than the specialized applications designed to solve specific range of goals.

Our prototype simulates a teleconference planning process between two people, and after a certain amount of straightforward work it will be able to connect two real people using audio and video streams. This demo version was used to polish the overall design of the planning system and resulted in a clear separation of the planning and pebble layers. Several major concepts were investigated and the best structure for them was suggested: it is advantageous to use feature set representation of goals, as well as use feature sets for the technique interface. Contexts are organized in hierarchical structures which are used for looking up distributed objects, such as techniques, pebbles, and hosts. The lookup algorithms can be easily modified by local context policies.
It was discovered that the direct push approach for reporting execution failures and environment changes works faster for goal tree reevaluations but requires a special infrastructure. In addition, the effect of the distribution of the planning system through the use of Java RMI was studied, and problem spots for network use, remote exception handling, and goal tree visualization were identified.

Thus, our goal-oriented planning system seems to be an indispensable tool (along with speech and face recognition technologies) for the simplification of the human interface for Oxygen platform.

11. Appendix

11.1. Sample Techniques for the Teleconference Demo

```java
// First Technique: set up both audio & video connections:
add (new Technique () {
    { name = "Full Audio/Video"; goalName = "Teleconference"; }
    int minimumBandwidth = 128000;
    void initialize () throws ExpansionException {
        Object src = getGoal().getSrc();
        Object dest = getGoal().getDest();
        satisfy (new Goal("ConnectAudio", src, dest));
        satisfy (new Goal("ConnectVideo", src, dest));
    }
});

// Second technique: audio only.
add (new Technique () {
    {name = "Audio Only"; goalName = "Teleconference"; }
    Choice audio;
    void initialize () throws ExpansionException {
        Object src = getGoal().getSrc();
        Object dest = getGoal().getDest();
        audio = satisfy (new Goal("ConnectAudio", src, dest));
    }
    void evaluate () {
        // Audio is only 10% satisfactory:
        setSatisfaction (audio.evaluateNode() / 10);
    }
});

/***
// Techniques for the ConnectAudio goal:
// Establish a bidirectional Audio link between people.
***/
add (new Technique () {
    { name = "Internet Audio"; goalName = "ConnectAudio"; }
});
```
void initialize () throws ExpansionException {
    Object src = getGoal().getSrc();
    Object dest = getGoal().getDest();
    satisfy (new Goal ("AudioLink", src, dest));
    satisfy (new Goal ("AudioLink", dest, src));
}

/***
// Second approach: LAN, uncompressed audio.
***/
add (new Technique () {
    { name = "100MBit LAN uncompressed"; goalName = "AudioLink"; }
    int requiredBandwidth = 128000;
    Choice audioSrc, audioDest, connection;
    Pebble aSrc, aDest;
    Connection conn;
    void initialize () throws ExpansionException {
        Object src = getGoal().getSrc();
        Object dest = getGoal().getDest();
        audioSrc = satisfy (new Goal ("getAudioSrc", src));
        audioDest = satisfy (new Goal ("getAudioDest", dest));
        connection = satisfy (new Goal ("getConnection", src, dest));
    }
    void evaluate () {
        // Diminish satisfaction slightly, to reflect cost of net connection:
        setSatisfaction (0.99 * Math.min (audioSrc.evaluateNode(),
                                           audioDest.evaluateNode()));
    }
    void execute () throws ExecutionException, RemoteException {
        conn = (Connection) connection.getImplementation();
        aSrc = (Pebble) audioSrc.getImplementation();
        aDest = (Pebble) audioDest.getImplementation();
        aSrc.connect (conn.getIn());
        conn.getOut().connect (aDest);
    }
    void reset () throws RemoteException {
        aSrc.reset();
        conn.getOut().reset();
    }
});

/***
// Techniques for the ConnectVideo goal:
// Establish a bidirectional Video link between people.
***/

// First technique: 2 live VideoLinks, one in each direction:
add (new Technique () {
    { name = "Simple unidirectional links"; goalName = "ConnectVideo"; }
    Choice forward, reverse;
    void initialize () throws ExpansionException {
        Object src = getGoal().getSrc();
        Object dest = getGoal().getDest();
forward = satisfy (new Goal ("VideoLink", src, dest));
reverse = satisfy (new Goal ("VideoLink", dest, src));
}
void evaluate () {
  setSatisfaction ((forward.evaluateNode() + reverse.evaluateNode()) / 2);
}

add (new Technique () {
  { name = "LAN Compressed"; goalName = "VideoLink"; }
Integer requiredBandwidth = new Integer (500000);
Choice videoSrc, videoDest, connection;
Pebble videoIn, videoOut, send, rcv, decode, compress;
Person src, dest;
void Initialize () throws ExpansionException {
  src = (Person) getGoal().getSrc();
  dest = (Person) getGoal().getDest();
  videoSrc = satisfy (new Goal ("getVideoSrc", src));
  videoDest = satisfy (new Goal ("getVideoDest", dest));
  connection = satisfy (new Goal ("getConnection", src, dest, requiredBandwidth));
}
void evaluate () {
  // Diminish satisfaction slightly, to reflect cost of net connection:
  setSatisfaction (0.99 * Math.min (videoSrc.evaluateNode(),
    videoDest.evaluateNode()));
}
void execute () throws ExecutionException, RemoteException {
  videoIn = (Pebble) videoSrc.getImplementation();
  videoOut = (Pebble) videoDest.getImplementation();
  Connection conn = (Connection) connection.getImplementation();
  send = conn.getIn();
  rcv = conn.getOut();
  decode = videoOut.getHost().installPebble ("Video Decode");
  compress = videoIn.getHost().installPebble ("Video Compress");
  videoIn.connect (compress);
  compression.connect (send);
  rcv.connect (decode);
  decode.connect (videoOut);
}
void reset () throws RemoteException {
  videoIn.reset();
  compress.reset();
  rcv.reset();
  decode.reset();
}
});

/**
// ... or fake it with a still picture:
***
add (new Technique () {
  { name = "Still Picture"; goalName = "VideoLink"; }
Choice videoSrc, videoDest;
Pebble vSrc, vDest;
void initialize () throws ExpansionException {
  Object src = getGoal().getSrc();
  ...
Object dest = getGoal().getDest();
videoSrc = satisfy (new Goal("getStillPix", src));
videoDest = satisfy (new Goal("getVideoDest", dest));
}
void evaluate () {
    setSatisfaction (Math.min (videoSrc.evaluateNodeo,
                             videoDest.evaluateNodeo)) / 3);
}
void execute () throws ExecutionException, RemoteException {
    vSrc = (Pebble) videoSrc.getImplementation();
    vDest = (Pebble) videoDest.getImplementation();
    vSrc.connect (vDest);
}
void reset () throws RemoteException {
    vSrc.reset();
});

/**
   // Access to audio, video from foreign regions:
   ***/
add (new Technique () {
    { name = "External Context"; goalName = "getAudioDest"; }
void evaluate () {
    Person person = (Person) getGoal().getArg0();
    try {
        Pebble port = person.get (Host.AUDIODEST);
        if (port == null) {
            fail (Util.toString (person) + " isn't accepting incoming audio");
            return;
        }
        setImplementation (port);
        setSatisfaction (1);
    } catch (RemoteException e) {
        fail (person + " is not accessible");
    }
}
});

add (new Technique () {
    { name = "External Context"; goalName = "getAudioSrc"; }
void evaluate () {
    Person person = (Person) getGoal().getArg0();
    try {
        Pebble port = person.get (Host.AUDIO_SRC);
        if (port == null) {
            fail (Util.toString (person) + " doesn't have a microphone nearby");
            return;
        }
        setImplementation (port);
        setSatisfaction (1);
    } catch (RemoteException e) {
        fail (person + " is not accessible");
    }
}
add (new Technique () {
    { name = "External Context"; goalName = "getVideoDest"; }
    void evaluate () {
        Person person = (Person) getGoal().getArgO();
        try {
            Pebble port = person.get (Host.VIDEO_DEST);
            if (port == null) {
                fail (Util.toString (person) + " isn't accepting incoming video");
            }
        }
        catch (RemoteException e) {
            setImplementation (port);
            setSatisfaction (1);
        }
    }
});

add (new Technique () {
    { name = "External Context"; goalName = "getVideoSrc"; }
    void evaluate () {
        Person person = (Person) getGoal().getArgO();
        try {
            Pebble port = person.get (Host.VIDEO_SRC);
            if (port == null) {
                fail (Util.toString (person) + " isn't in front of a camera");
            }
        }
        catch (RemoteException e) {
            fail (person + " is not accessible");
        }
    }
});

add (new Technique () {
    { name = "LAN"; goalName = "getConnection"; }
    Pebble bridge, send, rcv;
    void evaluate () throws RemoteException {
        Person src = (Person) getGoal().getSrc();
        Person dest = (Person) getGoal().getDest();
        try {
            send = src.get (Host.NET_SEND);
        }
        catch (RemoteException e) {
            fail (src + " is not accessible");
            return;
        }
}
try {
    rcv = dest.get (Host.NET_RCV);
} catch (RemoteException e) {
    fail (dest + " is not accessible");
    return;
}
if (send == null) {
    fail (Util.toString (src) + " isn't network connected.");
    return;
}
if (rcv == null) {
    fail (Util.toString (dest) + " isn't network connected.");
    return;
}
Integer arg2 = (Integer) getGoal().getArg2();
if (arg2 != null) {
    int requiredBandwidth = arg2.intValue();
    if (send.getMaxBandwidth() < requiredBandwidth ||
        rcv.getMaxBandwidth() < requiredBandwidth) {
        fail ("Insufficient network bandwidth.");
        return;
    }
}
// If a bridge is necessary, allocate one:
if (send.getOutType() != rcv.getInType()) {
    // arbitrarily put bridge on the src host
    bridge = src.getHost().installPebble ("Bridge");
}
setSatisfaction (1.0);
}
void execute () throws ExecutionException, RemoteException {
    if (bridge == null) {
        send.connect (rcv);
    } else {
        send.connect (bridge);
        bridge.connect (rcv);
    }
    setImplementation (new Connection (send, rev));
}
void reset () throws RemoteException {
    send.reset ();
    if (bridge != null) {
        bridge.reset();
    }
}
}

// Everyone has a still picture cached in their Person object:
add (new Technique () {
    { name = "Still Picture"; goalName = "getStillPix"; }
    Pebble pixSrc;
    void evaluate () throws RemoteException {
        setSatisfaction (1.0);
        // put pixSrc on src host
pixSrc = ((Person) getGoal().getSrc().getHost().
    installPebble("pixSrc");
}
void execute() throws ExecutionException {
    setImplementation (pixSrc);
}

// if the video is not possible in one direction, do not give up on the other
// direction
add (new Technique () {
    { name = "No Link"; goalName = "VideoLink"; }
    void evaluate() {
        setSatisfaction (0.0);
    }
});

12. References

1. Project Oxygen (http://oxygen.lcs.mit.edu/)


    01, University of Washington, Seattle, WA, October 8, 1998.

4. S. A. Wolfman, D. S. Weld. Combining Linear Programming and Satisfiability

5. D. Saff. Personal communication.

6. Java RMI Tutorial (http://java.sun.com/docs/books/tutorial/rmi/)

7. Java Language Specification
    2986)


9. Resource Description Framework (http://www.w3.org/RDF/)