IWAKI INTERACTION MANAGER 0.1
A MANUAL

MAXIM MAKATCHEV

August 13, 2013

1 INTRODUCTION

Iwaki Interaction Manager is an open-source dialogue manager that is not limited to dialogue. It is most closely related to the *SharedPlan* theory of discourse [Grosz and Sidner, 1990; Lockbaum, 1998], and its implementation, COLLAGEN [Rich and Sidner, 1998]. From the interaction perspective, it is designed to enable:

- Multi-modal interaction. Iwaki actions can be be interpreted by action generators of any nature.
- Multi-party interaction. Iwaki has the functionality to keep the context related to each individual interactor.
- Flexible turn taking and mixed initiative. As a dialog manager, Iwaki can take initiative in turn taking based on the system state, such as time. Similarly, Iwaki can process a user utterance produced at any time.
- Flexible discourse structure. Iwaki can respond to the user's utterance that is out of the current context.

From the interaction designer's perspective, Iwaki differs from some other dialogue managers in that:

- Interaction designer and content developer do not need to write code. Iwaki scripts are XML that can be generated via web forms or by hand.
- Iwaki does not make you choose between information-state and a script-based dialogue management. To an extent, you can have both.
- Iwaki is designed to play well with other developing standards for interaction design, such as BML.
- Iwaki is soft real-time. Iwaki scripts can be time-sensitive, hence it likes to be called periodically and on time.

From the software architecture prospective, Iwaki is a C++ library that can be called periodically from your interaction module to update its state and process inputs and outputs. In a complex application, however, to ensure timely periodic calls to the interaction manager, Iwaki library would likely be wrapped into a separate executable and interface with the rest of the system via an interprocess communication.

1.1 RELATED WORK

Collaborative agent-based approach to dialogue management is only one of several approaches to dialogue management (see an overview by Bui [2006]). Previously, the definitive implementation of the *SharedPlan* model [Grosz and Sidner, 1990] of collaborative discourse was the Collagen collaboration manager [Rich and Sidner, 1998], which was proprietary. An open source successor of Collagen, called Disco [Rich and Sidner, 2012], is currently under active development. An extension of SharedPlans that explicitly represents the task structure, called Collaborative Problem-Solving [Blaylock and Allen, 2005], has been used in another proprietary dialogue manager, SAMMIE Becker et al. [2006]. In the development of the IM, we follow the main ideas presented in Collagen and Disco and adapt them for our specific needs.

1.2 OVERVIEW

Similar to Collagen and Disco, IM creates and maintains a dialogue tree, called *interaction tree* that represents the current state of the interaction. Another component of the state is a set of global objects, called *atoms*, with variable *slots*. The global atoms are visible and writable from every node of the dialogue tree. The nodes of the dialogue tree are the instantiated *recipes*.

At present, Iwaki has no separate task structure. According to the collaboration discourse theory, a dialogue is viewed as a hierarchy of tasks, and an utterance can contribute to a task in one of the three ways: (1) provide a needed input, (2) select a new task or subtask to work on, or (3) select a recipe to achieve the task. Like Collagen and Disco, IM extends this interpretation paradigm to include inputs of other modalities.

The interaction tree-growing process is a production-like system (c.f. CLIPS expert system [Giarratano and Riley, 1998]) that attempts to satisfy goals by backchaining on rules. All the inputs and state changes are interpreted in the context of the current interaction tree. According to the joint plan theory of interaction, every participant of the interaction contributes to building the joint plan. Generally, each participant may have their own view of the joint plan, distinct from others. Iwaki considers only one view of the joint plan, which can be considered the robot's point of view.

The knowledge that comes from outside of the interaction manager (facts) is represented as a set (actually, a conjunction) of *atoms*, where each atom is a partially grounded predicate.

IM *recipes* have a function similar to Disco and Collagen recipes, and correspond to the rules of a production system. A recipe consists of a precondition, a body and a postcondition. A precondition is a disjunctive normal form (DNF) of partially grounded atoms. A body

of the recipe is an ordered sequence of *actions*, *assignments*, or *goals*. The postcondition of a recipe is an assignment.

The are several ways in which Iwaki interaction manager differs from a typical production system:

- Items in the body of a recipe (corresponding to the right-hand side of a production rule) are executed in order (sequentially by default). Generally, the execution proceeds to the next item when execution of the previous item is successfully completed.
- There is a mechanism to control the flow of execution depending on the return status of actions. For example, if a time-sensitive action, such as saying "goodbye" to a departing user, has been aborted by the executor, for some reason, Iwaki may need to purge the rest of the farewell recipe. If that was a greeting, Iwaki may retry sending it to the behavior executor, or skip it and move to the next action in the greeting recipe.
- There is a mechanism for timing out of items in the body of a recipe.
- There is a whilecondition, failure of which triggers purging of the recipe.

Iwaki interaction tree interface implements the three ways of interpretation of an input specified by collaborative discourse theory. Notably, Iwaki allows interleaved recipe execution, resulting in simultaneous multi-topic dialogues. However, at the present state, the interaction manager does not implement many of the features of Collagen and Disco, such as a *focus stack*.

1.3 APPLICATIONS

Iwaki interaction manager has been originally developed to control interaction of social robots at the Robotics Institute of Carnegie Mellon University. It powers such robots as Gamebot, the scrabble-playing trash-talking robot, and Hala 2, the bilingual robot receptionist at CMU Qatar [Simmons et al., 2011]. It has also been used as a rather advanced soundboard to generate dialogue for radio show sidekick characters. The soundboard application code is provided with Iwaki distribution as an example.

1.4 LICENSE

For a copy of the GNU General Public License see http://www.gnu.org/licenses/.

2 QUICK START

This chapter's goal is to quickly get the reader going with the Iwaki installation and the example soundboard application.

2.1 INSTALLING ON UNIX

2.1.1 *Prerequisites*

In addition to the usual packages necessary to build a C++ executable on a Linux platform you will need the following libraries:

- `ncurses` library (available from Ubuntu repositories).
- `RE2` regular expression library. Install it from `http://code.google.com/p/re2`

Furthermore, the Soundboard application plays sounds using gstreamer, so make sure it is installed and runnable via via `gst-launch-0.10` command.

2.1.2 *Install Iwaki*

In the root of the Iwaki distribution directory, which we will refer to as `PROJECTDIR`, examine the file CMakeLists.txt. Make sure it includes the correct location of your RE2 library and header files.

To build the binaries in a directory outside of the source tree, type

```
mkdir build
cd build
cmake ..
make
```

2.2 RUNNING SOUNDBOARD APPLICATION

The soundboard executable should be created in `PROJECTDIR/build/soundboard/bin`. You can run it from `PROJECTDIR/build` by typing the following command line (make sure to replace PROJECTDIR with the actual location of the root of the installed Iwaki distribution).

```
./soundboard/bin/soundboard -t 0.1 -d DEBUG4 -l log1 -p ../
    soundboard/scripts -i initialize_im.georgi.xml -s PROJECTDIR/
    soundboard/sounds -x
```

The command line options specify the following:

`-h` Prints command line options information

`-d level` One of the following debug levels: `ERROR`, `WARNING`, `INFO`, `DEBUG`, `DEBUG1`, `DEBUG2`, `DEBUG3`, `DEBUG4`

`-t timer_period` Specifies the period in seconds between calls to Iwaki update.

`-l path/log_file_prefix` If present, this option directs all the debug output to the file prefixed with `log_file_prefix`.

`-x` If present, this option enables the text-based user interface displayed in the current terminal window.

`-p path` Path to IM scripts directory. This directory should contain init file.

`-i init_file` If present, overrides the default init file name, initialize_im.xml.

`-s path to sound files` Absolute path to the sound file directory.

If Soundboard has successfully started you should hear a hearbeat sound at random integer intervals between 1 and 4 seconds. Type "h" or "how" and press enter to greet the Soundboard character, you should hear a response.

3 CONTENT REPRESENTATION IN IWAKI

3.1 INITIALIZATION FILE

Iwaki initialization file is located in the root of scripts directory, and by default is named `initialize_im.xml`. The default init file name can be changed by `-i init_file` command line option.

The initializer file has three sections: list of triggerable recipes, list of recipe files and list of action files—see the example below.

```
<?xml version="1.0" encoding="US-ASCII"?>

<iminit>
  <!-- these recepies are always ready to be triggered by im
      itself-->
  <triggerables>
    <recipe name="how_are_you" max_instances="1"/>
    <recipe name="how_are_you_and_you" max_instances="1"/>
    <recipe name="heartbeat" max_instances="1"/>
  </triggerables>


  <recipefiles>
    <file>georgi/how_are_you_and_you.xml</file>
    <file>georgi/how_are_you.xml</file>
    <file>georgi/heartbeat.xml</file>
  </recipefiles>


  <actionfiles>
    <file>defaults.xml</file>
  </actionfiles>
</iminit>
```

3.2 RECIPES

IM is a production system, with rules specified as recipes. A recipe consists of the following elements:

- XML header:

  ```
  <?xml version="1.0" encoding="US-ASCII"?>
  ```

- A unique recipe name:

  ```
  <recipe name="heartbeat">
  ```

- A precondition, which is either a Formula (a disjunction of a few conjunctions), or a set of Atoms corresponding to a single conjunction:

```
<!-- preconditions: unification and bindings -->
<precondition>
    <atom quantifier="exist">
      <!-- object type and subtype -->
      <slot name="type" val="im"/>
      <slot name="subtype" val="globals"/>
      <!-- arguments -->
      <slot name="prev_beat_time" rel="<" val="\$_im_time-
          randi(1,5)" type="number"/>
      <!--bindings -->
      <slot name="time" rel="bind" var="_im_time"/>
      <slot name="this" rel="bind" var="the_globals_atom"/>
    </atom>
  </precondition>
```

- A whilecondition, which is either a Formula (a disjunction of a few conjunctions), or a set of Atoms corresponding to a single conjunction:

```
<!-- purge the recipe when this condition fails -->
<whilecondition>
  <atom>
    <slot name="present" val="true"/>
    <slot name="this" var="present_user_atom"/>
  </atom>
</whilecondition>
```

  Not that wilecondition can be empty, as in the `heartbeat` recipe.

- A body of the recipe, which is a sequence of one of the following elements: an assignment, an action, a goal. By default the steps are performed sequentially and conditionally on the successful execution of the preceding steps.

```
 <body order="sequence">
    <assignment>
      <atom>
        <slot name="prev_beat_time" val="$_im_time" type="
            number"/>
        <slot name="this" var="the_globals_atom"/>
      </atom>
    </assignment>

    <action name="generate_utterance" actor="generator"
        action_space="speech">
      <roboml:args>
        <arg name="utterance_file" value="georgi/heartbeat.
            ogg" type="string"/>
      </roboml:args>
```

```
    </action>

  </body>
```

- An assignpost, which is an assignment that is to be performed upon the completion of the recipe. This assignment are equivalent to the assignments in the end of the recipe body, except for they are also used in matching the recipe against the currently active goal.

```
<!-- set right after execution ends -->
  <assignpost>
    <atom>
    <!-- set object which name is equal to the one stored in
         var -->
      <slot name="this" var="present_user_atom"/>
      <slot name="q2u_how_are_you_handled" val="true"/>
    </atom>
  </assignpost>
```

3.3 ACTION DEFINITION FILES

Actions are the outputs of the Interaction Manager. They consist of a list of arguments, sequence of datablocks, and elements defined by the Behavior Markup Language [BML, 2012].

Here is an action with 4 arguments:

```
<bml name="generate_utterance">
  <roboml:args>
    <arg name="utterance_string" default="_NO_VALUE_" type="
        string"/>
    <arg name="utterance_token" default="_NO_VALUE_" type="string
        "/>
    <arg name="utterance_file" default="_NO_VALUE_" type="string"
        />
    <arg name="test_arg" default="0" type="number"/>
  </roboml:args>
</bml>
```

3.4 DEFAULT ATOMS FILE

4 RECIPES

In Section 3.2, we outlined the structure of a recipe. In this chapter we will describe each of the recipe components in detail. You will know how to write a recipe to respond to a particular input, and how to implement control flow, including conditionals via backchaining.

4.1 PRECONDITIONS

4.2 BODY

A body of the recipe consists of a sequence of *body elements*. A body element is an *assignment*, an *action*, or a *goal*. At present (version 0.1), body elements are executed sequentially. Action and goal body elements are different from the assignment in that they may take undetermined amount of time. To allow for the execution to continue even if an action or a goal takes too long time to complete, action tag has an optional timeout attribute. Actions are different from both assignments and goals in that their execution may return a status different from the successful *completed* status. Returned action status can be used to specify the control flow for the rest of the recipe, such as moving to the next body element, moving to the assignpost, or moving to the end (purging the recipe).

4.2.1 *Assignment*

4.2.2 *Action*

4.2.3 *Goal*

A goal specifies the set of recipes that can be used to backchain on it. There are two way to define a goal. First, via a formula that must be satisfied by the child recipe. Second, by explicitely listing the names of potential child recipes. In the latter case, the formula is optional. We refer to a goal without a formula as an *empty goal*.

A goal with a formula

An example of a goal with a formula is show below:

```
<goal recipe_name="any">
  <atom>
    <slot name="q2u_how_are_you_handled" val="true"/>
    <slot name="this" var="present_user_atom"/>
```

```
    </atom>
  </goal>
```

During the preprocessing stage, for each of the goals, Iwaki idetifies recipes whose postconditions, once executed, will satisfy the goal's formula. These recipes are selected from those listed in the *recipe_name* attribute. If the *recipe_name* attribute is empty, or is equal to 'any', then all recipes loaded via init file are potential candidates.

4.2.4 *An empty goal*

```
<goal recipe_name="q2u_how_are_you_answer_yes,
    q2u_how_are_you_answer_no"/>
```

An empty goal does not include a formula. Potential child recipes are specified by *recipe_name* attribute. If *recipe_name* is left empty or set to "any" a warning will be issued at the preprocessing stage, since that means that any recipe can be a potential child.

4.2.5 *Control flow with the goals*

When execution reaches the goal elemenet of a recipe's body, the further control flow depends on whether the goal contains a formula or not, and whether the *forced* attribute of goal tag is set to "true" or not.

- If the goal's formula is not empty *and* the goal's *forced* attribute is not set to "true", the formula will be checked for satisfiability against current global state of the system. If the formula is already satisfied, the execution flow moves to the next body element (or the postcondition if the goal was the last element of the body). If the formula is not satisfied, backchainable recipes will be attempted, in order of their priority, to be backchained upon by matching them against the goal and checking if their preconditins are satisfied.
- If the goal's formula is not empty *and* the goal's *forced* attribute is set to "true," then the goal's formula is *not* checked on satisfiability by current state, and the attempt to backchain is forced independently on whether the goal's formula is already satisfied.
- If the goal's formula is empty, then the attempt to backchain is made on the backchainable recipes specified in the goal's *recipe_name* attribute. This can be thought as a forced backchaining on a formula that is already satisfied, since an empty formula is true by the definition.

Why would we want the forced backchaining on a non-empty goal? Consider the following example. We would like to backchain on a set

of recipes that are conditioned an uninstantiated user atom and execute some action for that particular user, once it gets instatiated. The parent recipe (the one that contains the goal) may want to constrain the children recipes to be instantiated (and their preconditions to be checked against) a particular user atom. In that case, the goal may look as follows:

```
<goal recipe_name="q2u_how_are_you_answer_yes,
    q2u_how_are_you_answer_no" forced="true">
  <atom>
    <slot name="this" var="present_user_atom"/>
  </atom>
</goal>
```

The postconditions of *q2u_how_are_you_answer_yes*, *q2u_how_are_you_answer_no* recipes may be something like this:

```
<assignpost>
  <atom>
    <slot name="this" var="present_user_atom_of_child_recipe"/>
  </atom>
</assignpost>
```

Since the atom in the goal's formula in this example will match any atom of postcondition (because *this* slot of postconiditon is always uninstantiated before the actual backchained recipe is loaded), more slots may be necessary if there is an ambiguity between the goal's and postcondition's atoms.

4.3 POSTCONDITIONS

BIBLIOGRAPHY

Tilman Becker, Nate Blaylock, Ciprian Gerstenberger, Ivana Kruijff-korbayová, Andreas Korthauer, Manfred Pinkal, Michael Pitz, Peter Poller, and Jan Schehl. Natural and intuitive multimodal dialogue for in-car applications: The sammie system. In *In Proceedings of the ECAI Sub-Conference on Prestigious Applications of Intelligent Systems (PAIS 2006), Riva del Garda*, 2006. (Cited on page 2.)

Nate Blaylock and James Allen. A collaborative problem-solving model of dialogue. In *In Proceedings of the SIGdial Workshop on Discourse and Dialog*, pages 200–211, 2005. (Cited on page 2.)

BML. Behavior Markup Language 1.0 Standard. `www.mindmakers.org/projects/bml-1-0/wiki/Wiki`, July 2012. Accessed July 2, 2012. (Cited on page 9.)

Trung H. Bui. Multimodal dialogue management—state of the art. Technical report, Human Media Interaction Department, University of Twente, The Netherlands, January 2006. (Cited on page 2.)

Joseph C. Giarratano and Gary D. Riley. *Expert Systems: Principles and Programming*. Course Technology, 1998. (Cited on page 2.)

B. J. Grosz and C. L. Sidner. Plans for discourse. In P. Cohen, J. Morgan, and M. Pollack, editors, *Intentions in communication*, pages 417–444. MIT press, 1990. (Cited on pages 1 and 2.)

K. E. Lockbaum. A collaborative planning model of intentional structure. *Computational linguistics*, 24(4):525–572, 1998. (Cited on page 1.)

Charles Rich and Candace L. Sidner. Collagen: A collaboration manager for software interface agents. *Journal of User Modeling and User-Adapted Interaction*, 8:315–350, 1998. (Cited on pages 1 and 2.)

Charles Rich and Candace L. Sidner. Using collaborative discourse theory to partially automate dialogue tree authoring. In *Proc. Int. Conf. on Intelligent Virtual Agents*, 2012. (Cited on page 2.)

R. Simmons, M. Makatchev, R. Kirby, M.K. Lee, I. Fanaswala, B. Browning, J. Forlizzi, and M. Sakr. Believable robot characters. *AI Magazine*, 32(4):39–52, 2011. (Cited on page 3.)