A Real-Time Architecture for Conversational Agents

by

Bahador Nooraei

A Thesis
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
In partial fulfillment of the requirements for the
Degree of Master of Science
in
Computer Science
July 2012

APPROVED:

Professor Charles Rich, Thesis Advisor

Professor Candace L. Sidner, Thesis Co-Advisor

Professor George Heineman, Thesis Reader

Professor Craig Wills, Head of Department

The door was the way to… to… The Door was The Way. Good. Capital letters were always the best way of dealing with things you didn't have a good answer to.

Douglas Adams, *Dirk Gently's Holistic Detective Agency*

Abstract

Consider two people having a face-to-face conversation. They sometimes listen, sometimes talk, and sometimes interrupt each other. They use facial expressions to signal that they are confused. They point at objects. They jump from topic to topic opportunistically. When another acquaintance walks by, they nod and say hello. All the while they have other concerns on their mind, such as not missing the meeting that starts in 10 minutes.

Like many other humans behaviors, these are not easy to replicate in artificial agents. In this work we look into the design requirements of an embodied agent that can participate in such natural conversations in a mixed-initiative, multi-modal setting. Such an agent needs to understand participating in a conversation is not merely a matter of sending a message and then waiting to receive a response both partners are simultaneously active at all times. This agent should be able to deal with different, sometimes conflicting goals, and be always ready to address events that may interrupt the current topic of conversation.

To address those requirement, we have created a modular architecture that includes distributed functional units that compete with each other to gain control over available *resources*. Each of these units, called a *schema*, has its own sense-think-act cycle. In the field of robotics, this design is often referred to as behavior-based or schema-based. The major contribution of this work is merging behavior-based robotics with plan-based human-computer interaction.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. IIS-1012083. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Contents

1 Introduction **1**

1.1 A Motivating Scenario 4

1.2 A Note on Implementation 7

1.3 Related Work 8

2 Use Cases and Requirements **11**

2.1 Interaction Use Cases 11

2.2 Activity Management 13

2.3 Other Design Goals and Challenges 15

3 Architecture Fundamentals **17**

3.1 Overview 17

3.2 Resources 18

3.3 Behaviors 19

3.3.1 Primitive Behaviors 19

3.3.2 Compound Behaviors 19

3.4 Realizers 20

3.4.1 Primitive Realizers 20

3.4.2 Compound Realizers 21

3.5 Perceptors 21

3.6 Schemas 22
3.7 Schema Manager 23
3.8 Arbitrator 23

4 Behavior Arbitration 25

4.1 Arbitration Parameters 28
4.2 Fuzzy Arbitration Rules 30
4.2.1 Maintaining The Rule Set 34

5 Additional Built-in Components 37

5.1 Engagement Schema 37
5.2 Activity Starter Schema 38
5.3 Dialog-Based Schemas 38
5.4 Plugins and Behavior Builders 39

6 Demonstration and Evaluation 40

6.1 Sample Scenario Revisited 40

7 Conclusion and Future Work 48

A Fuzzy Arbitration Details 55

A.1 Input and Output Variables 55
A.2 Fuzzy Rules 57

List of Figures

2.1 An activity plan 14

3.1 Architecture overview 17
3.2 Decision making 21

4.1 Fuzzy set memberships for *DueIn* parameter 31
4.2 The effect of changing the *Due In* parameter in the second activity. . 34
4.3 The effect of changing the *Specificity* parameter in the second activity, when (a) it is not a new activity, (b) it is a new activity. 35
4.4 The effect of changing the *TimeActive* parameter in the first activity, when (a) it is a container, (b) it is not a container. 36
4.5 A Fitnesse test 36

Chapter 1

Introduction

As computers are becoming more and more part of our daily activities, the need grows for more *natural* interfaces, that enable a wider range of people to use computer systems without any specialized training, grows too. A good deal of research in artificial intelligence is towards an ultimate goal of making a computer behave more like a person we are conversing with, and less like a device with its own cryptic set of rules of operations. In other words, we want to adapt computers to humans, instead of forcing humans to adapt to computers. Embodied conversational agents are one form of such intelligent interfaces that has been the subject of many research projects. In a conversational agent, the task at hand is achieved through a dialog with the user. At the current state of the art, the content of this dialog is mostly manually authored, but in some systems, higher-level semantics can be encoded in a way that allows the system to automatically generate some utterances at the appropriate moments. Furthermore in traditional conversational agents the structure of the conversation is rigidly controlled by *turns*, i.e., one side of the conversation (the agent or the person) has the floor, says something, and then the floor is given to the other side. In contrast, in a natural conversation between people, *both* participants are active all the time, sending verbal and non-verbal signals back and forth simultaneously. This is what Clark [13] calls *simultaneity*, an important property of face-to-face conversations between people. Being active at all times, goes hand in hand with the ability to recognize, respond to, and generate non-verbal behavior. This ability is considered a characteristic feature of embodied conversation agents [11].

In this work, we focus on two key features of natural conversation, that we want to achieve in conversational agents: *multimodality* and *timing*.

Multimodality People use a combination of speech, gaze, hand gestures, head nods, etc. to convey our ideas. In many cases, different non-verbal channels are used to emphasize or clarify aspects of the information communicated via speech, e.g., pointing at an object when talking about it. In other cases, the signal sent via a non-verbal channel may have a communicative objective of its own, making its understanding or generation vital to an effective interaction.

Timing Consider a person saying this sentence with accompanying pointing gestures: "put that {*pointing at a book*}, there {*pointing at a table*}." There is a specific synchronization required between the pointing gestures and the words "that" or "there," in this example. This synchronization requires correct timing on the order of milliseconds. There are other time scales involved in natural conversations. One is the time scale of seconds and minutes, the realm of topic and activity management. At this scale we decide on what needs to be discussed about current activities, which activity we should focus at any moment, and what other activities it is possible to start. Finally, there is the time scale of days and weeks, that governs the long-term relationship between people. This scale is the subject of related work by Coon [14].

What we observed was that a conversational agent has a number of different *goals*, from high-level topic management to low-level reactive behavior such as looking at a sudden movement in the background. Viewed from this perspective, it is much like many robotic systems that do not necessarily involve any conversation. This led us to take ideas from the world of mobile robots that navigate in the real world and combine them with plan-based dialog techniques that involve verbal and non-verbal signals. The resulting architecture we have developed, a highly distributed design following the schema theory ideas of Arbib [2], achieves high levels of reactivity in real-time, and allows for independent authorship of different behaviors (plugins) for the system. As in other distributed designs there is a *behavior arbitration* component that selects the best course of action from the options that the behavioral schemas are suggesting at any point in time.

A novel concept our system supports is what we call *container* activities. Many social conversations occur *during* another activity. For example, people talk about many different things when they are at the dinner table, or when they are playing cards. Interestingly, in these situations people switch back and forth between talking about the "container" activity ("Could you hand me the salt shaker?") and the contained dialog ("So how did John react when his mother said that?"). The notion of container activities was a major force in driving us towards using a schema-based architecture, because schemas enable concurrent, isolated processes each following its own set of behaviral rules, and storing and updating relevant information.

In the rest of this chapter, we will present a sample scenario that motivates our system, and then review some background literature that contributed to our work. In Chapter 2 we delve into the details of the requirements for the architecture. Chapter 3 describes our implemented architecture in depth. The details of the behavior arbitration mechanism are discussed in Chapter 4.

1.1 A Motivating Scenario

This work is part of a larger research project with the goal of creating an always-on agent for social support of isolated older adults[1]. Our vision is a conversational virtual agent living in an apartment with an isolated senior. This framing of the problem forced us to think in very practical terms about a software system that could work reliably using today's technology. For example, we decided *not* to use speech recognition, and instead to use the kind of with a simple menu based interface that have been successfully used in projects previously [5]. This means that the authored dialogs in the system also include all potential responses from user (to be displayed as menu choices).

The user picks the desired menu on a touch-screen display. There is camera attached to the system, enabling the agent to detect a human face in front of it. The system is also equipped with a passive infrared (PIR) sensor to sense motion in its vicinity.

The scenario presented in this section demonstrates a range of phenomena we want the system to support, including container activities (in the form a card game), managing the flow of conversation by switching from topic to topic, interrupting the current task when something urgent comes up, and synchronized verbal and non-verbal behavior directed gaze. All these use cases will be covered in depth in Chapter 2.

To show the flow of the envisioned interaction better, in the following scenario we are only including the responses selected by the user, and not all the menu options rendered on the touch screen, except when needed to demonstrate something significant.

---

[1] http://web.cs.wpi.edu/~rich/always/

[Using the camera and PIR sensor, the system detects the person approaching the computer, and lights up the screen]

AGENT: Good Morning

USER: Good Morning

A: Did you sleep well?

U: Yes.

A: Are you feeling. . .

[Barge-in: The person touches the menu answer before the agent is done speaking, causing the agent to stop mid-sentence]

U: Yes.

A: Glad to hear it.

A: Do you want to know about today's weather?

U: Yes, how's today's weather?

A: It's quite warm, but it is going to rain in the afternoon.

U: Okay.

A: Do you want to hear about tomorrow too?

U: No, thanks.

A: Okay. What do you want to do?

[Menu options for calendar, rummy game, and baseball appear on the screen] U: What's on my calendar for today?

A: Not much. But you've got a trip to the aquarium at ten o'clock.

U: Okay.

A: What else do you want to do?

U: Let's play some rummy.

[The rummy game is displayed on the screen]

[They start playing.]

[There are dialogs about the game that we will not reproduce here. An example is when the agent discards a card:]

A: Okay, I'm done for now. So I discard *[looks at the card]* this one. *[looks at the user]* Now it's your turn.

[At some point the person leaves the computer temporarily. The agent starts an looking around, indicating that it no longer sees the user.]

[After three minutes, the person returns.]

A: I thought you don't want to continue the game.

[Menu options appear: "No, let's continue," and "Yes, let's finish this game later"]

U: No, let's continue.

[A few minutes later, the agent decides to bring up another topic while they continue playing the game]

A: You know, I've been thinking. Are you interested in finding a walking buddy?

[The person does not answer right away. After 30 seconds, the agents repeats the question]

A: I was saying, are you interested in finding a walking buddy?

U: What's that?

A: Walking with someone else is more fun. A walking buddy is a friend you walk with regularly.

U: Ok, how do we do that?

A: Well, we could try to find someone who lives in this building to go walking with you.

U: That sounds good.

⋮

[The game and the conversation goes on for a few more minutes, then at 9:45]

A: Oh! It's almost ten o'clock! It's time for you to go to the lobby for the aquarium trip.

U: Okay, thanks

A: See you later.

[The rummy game is not on the screen anymore]

U: Bye

[The screens dims]

1.2 A Note on Implementation

The architecture proposed in this document has been implemented to *a*) be instrumental in iterative design of the architecture; *b*) demonstrate the ideas in practice; and *c*) serve as the real-time control component of a relational agent, in a larger project of a relational agent for isolated older adults. However, the architectural ideas and the external behaviors exhibited in the current implementation should not be viewed as one and the same. That is to say, the core architectural concepts are to be considered, regardless of the specifics of the actuators, sensors, etc. that we are using. For instance, we made sure the system meets the necessary requirements for recognizing and generating engagement behavior by a few examples involving an on-screen card game. Even though the graphical agent used in our current implementation does not even have a good way of pointing at objects, it demonstrates that the proposed architecture is *capable* of incorporate ideas developed previously in our lab [19]. It shows that the combination of schemas, builders, and realizers, all of them concepts of the core architecture, can be used to implement the desired behavior with a reasonable amount of implementation effort.

In Chapter 3 we will cover the building blocks of the architecture such as resources, perceptors, behaviors, and schemas. It is worthwhile to note that for many, if not all, of those concepts there is a type-token distinction. Therefore, throughout this document we will not only talk about the *types* and general concepts, but also instances of those in our current implementation. Many of these instances are not discussed in detail since they do not add much to presentation of our ideas. For example, the current implementation uses a virtual agent designed by Relational Agents Group, and, accordingly, has Resources and Primitive Realizers specific to that. On the other hand, there are some specific instances that are essential parts of our design, and deserve entire sections devoted to them. For example, in section 5.2 we will talk in depth about a schema that follows high-level interaction plan handed down by a higher level component outside the scope of this architecture, and is in charge of starting available activities from that plan.

1.3 Related Work

Broadly speaking, this thesis is about building intelligent interactive agents. One of the classic approaches to intelligent agent design is the Belief-Desire-Intention (BDI) model [8, 16, 25]. In BDI agents, the agent maintains beliefs about the world, and has desires about what it wants to happen in the world. What causes the agent to act are the intentions it is currently holding. Shedding light on intentions as a distinct concept from desires and how desires give rise to intentions is the key contribution of the BDI as a philosophical model of human cognition. Expanding on BDI, various logical semantics, agent programming languages and implementations have been created. Sardiana and Padgham [32] is an example of current work on BDI-style programming languages and logical semantics.

Another classic approach to agent architecture in the robotics community is Brookss subsumption architecture. In his seminal paper [10], Brooks talks about requirements and goals that although presented for a mobile robot, apply to intelligent agents in general. For example, he points out that agents have multiple, sometimes conflicting, goals. Some goals are high level desires such as reaching a certain destination; some are lower level but necessary goals like avoiding obstacles. Pirjanian [24] classifies subsumption architecture as a priority-based arbitration model for behavior coordination. There are other classes of behavior coordination that have borrowed ideas from. Most notably, Arkins schema-based architecture [3, 4] which is a reactive/reflexive architecture based on primitive constructs called schemas. The concept of motor schemas and perception schemas is also motivated by brain theory and psychology. Good surveys of different behavior coordination mechanisms can be found in [24] and [33].

Our work is also built on the work of Rich & Sidner on managing human-computer collaboration using the SharedPlans theory of discourse [18, 28]. For task modeling and goal decomposition we will use Disco, a collaboration manager software package based on ANSI/CEA-2018 standard [26], and a successor to Collagen [30]. DiamondHelp, a task guidance system for home appliances built on top of Collagen, is a good demonstration of how a task-based interface can improve peoples interaction with computer systems [29].

From a natural language processing point of view, simultaneity requires an agent to incrementally process the speech input so that it can a) react to whatever has been said so far using verbal and non-verbal signals such as head nods, eye movements, and event interruptions, and b) form expectations about what is going to be uttered next by the person. This incrementality has been the subject of many agent designs, such as [1, 9]. Although linguistic incrementality on its own has no direct application in our project, these projects are motivated by the same phenomena that we are interested in. Therefore, their overall architecture is of interest to us. At the very least, we tried to create an architecture that is compatible with a potential incorporation of incrementality in the future.

Lemon et al. [22], and Bohus & Rudnicky [7], propose dialog management architectures with two layers running independently and asynchronously: a content layer and an interaction layer. The content layer is responsible for higher-level aspects such as goals, intentions, and conversational context. The interaction layer gets the content from the upper layer, and manages the interaction, using basic conversational strategies such as timing, turn-taking, and engagement behavior. Thorisson [34] created a comprehensive computational model for specifically for turn-taking. His work also emphasized the use of a multi-layered architecture, with three types of processing modules: perception, decision, and behavior [35]. This architecture was used in creating a conversational agent called Gandalf . More recently, Bohus & Horvitz [6] have developed a computational model for multi-party turn-taking, used in building virtual avatars, e.g., for playing trivia games.

Chapter 2

Use Cases and Requirements

In the chapter we cover use cases we wanted our system to support, plus other requirements and constraints on the design. It should be kept in mind that this conversational agent uses text-to-speech to talk to the person interacting with it, but does not use natural language understanding for accepting answers. Verbal answers are received via menus on a touch screen. It uses a camera to track the user's face or any other movements in the environment. We are using motion detection using passive infrared sensors in addition to the video camera. The specifics of how to gather robust sensory information is not a focus of this work.

First, we will enumerate all the interaction phenomena of interest. Next, we will talk about high-level activity management. Finally, we will discuss the requirements for making our architecture reusable and extensible.

2.1 Interaction Use Cases

Here is the list of use cases for a conversational agent that our architecture should support:

Engaging in dialog This is a base case for any conversation agent. Implementation note: we wanted the system to be able, but not limited, to using dialogs authored using the Disco framework.

Container activities These are activities, like a card game, that can be interleaved with some other activity on top of them. For example, during a card game, we want the agent and the person to be able to start another conversation, and then switch back and forth between playing (and talking) about the game, and that other conversation.

Barge-in We want the agent to be responsive enough that its speech can be interrupted by the user, and the agent responds accordingly. For example, when an utterance is being made, and we are passed a point that conveys the minimum necessary information to the user, we show the menus on screen. Now, the user can respond even before the agent is done uttering all the words, in which case it should stop, and accept the answer.

Connection events Appropriate engagement behaviors are fundamental to human interactions. Previous research [27, 19] has defined four types of *connection events*, including *directed gaze* (one person looking and optionally pointing at an object, and the other person looking at the same objects) and m*utual facial gaze*. The scenario presented in section 1.1 contains an instance of directed gaze during the card game.

Default gazing behavior The agent looks at the person interacting when there is no need to look somewhere else (e.g. during a directed gaze).

Appearance and disappearance of the user The system needs to respond believably when a user walks away from the computer, and later when he/she returns. For example, if the person walks away, it should stop talking, or after a while can start showing that it is “looking” to find the person. On return, it may just repeat the last utterance or even, as we saw in the scenario in Chapter 1, inquire about whether the person wants to stop the current activity.

Momentary distraction For example, when a movement happens in the background, the agent will glance up. This use case is a representative of the subtle actions that make an agent look more *aware* of the environment.

Event-based interruption This is support for any events that require an interruption of the current task. For example, when someone that the user wants to talk to, comes online in Skype. A special category of events is that which those that are clock-based, e.g. notifying the user about an upcoming calendar event.

2.2 Activity Management

One of the requirements in our architectural design has been the ability to follow a plan of discourse given by some higher level cognition. How this plan is created is the subject of Coon's master's thesis [14]. For our purposes here, we only need to understand what this plan contains, and how the architecture should follow it during a real-time interaction with the user. Figure 2.1 shows a simple plan containing three activities a_1, a_2, and a_3. These activities can be anything from a simple conversation about weather to reviewing the user's plans for the day.

The tree of activities represents different paths that the conversation may take during the course of the interaction. For example, considering the example in figure 2.1, at the beginning either the a_1 or a_2 can be started. Thus the sequence of activities can be $a_1 \rightarrow a_2 \rightarrow a_3$, $a_1 \rightarrow a_3 \rightarrow a_2$, or $a_2 \rightarrow a_3$. The idea behind

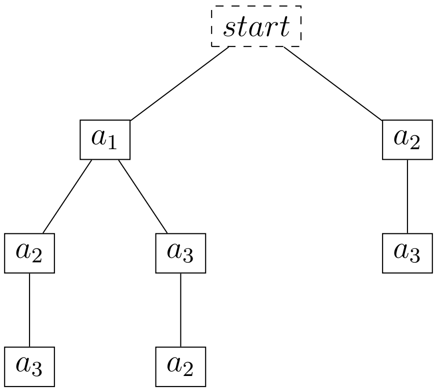


Figure 2.1: An activity plan

having these predefined courses of interaction is that the higher cognition of the agent, has a model of social relationship, for both a single session and over the weeks and months of acquaintance with the person. Using that knowledge and taking into account the utility of available activities, it can theoretically generates the most profitable courses of interaction that are likely to succeed. There can even be procedural preconditions for each activity, that are to be evaluated at run-time. For example, in our current implementation there's a quantity representing *social capital* gained during this session that increases according to the time spent in different activities. There may be some especially sensitive activity that requires a certain level of social capital in the session before it can be started, and this can be encoded as precondition of that activity. This was only mentioned as an example of what is possible to be encoded in the plan, but the details of how they are designed are outside the scope of the work presented in this document.

Container activities such as card games are treated differently and are not included in the plan mentioned above. These activities can interleave with other activities, and can be started at any time. They can be started at user's request or because there are no other options available (it can happen because of preconditions on activities in the plan, in that case the social capital gained from playing the game can help making those activities available). They certainly can be included in the plan like any other activity, but during our tests we saw that it only overcomplicates the plans, for example causing us to an extra branch at every node for starting the game. Recognizing the different nature of these activities helped us simplify the design considerably.

For completeness, it should be mentioned that there are event-based activities such as reminding about a calendar event, that naturally do not come from a predefined plan.

2.3 Other Design Goals and Challenges

Other than specific use cases specified above, there were a few broader goals and constraints that affected our design.

Content authoring As a practical way of expanding the system, and adding more capabilities and interaction scenarios over time, the system needs to support content providing plugins that can be added or removed to/from the system easily.

Independence from robotic platform Right now we did all our tests against a simple on-screen embodied agent. However, as an important goal we wanted to be able easily change this agent visualization, or even move to a physical, robotic platform. Therefore we need the reasonable abstractions that hide details of sensors and actuators, making the system easy to adapt to other platforms in the future.

Low overhead The system needs to be able to run on a single mainstream desktop computer, with all vision and sound processing also running on the same box.

Hence the core of the system should be designed in a way that does not make it *wait* excessively for other components and plugins.

Chapter 3

Architecture Fundamentals

3.1 Overview

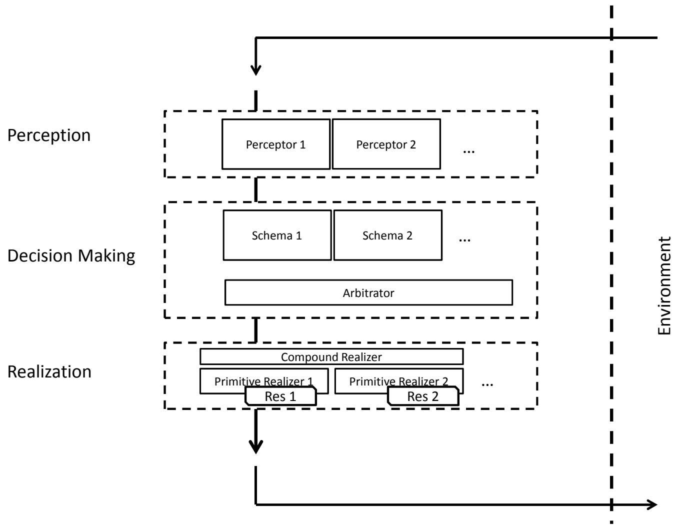


Figure 3.1: Architecture overview

In this chapter we introduce important terms, concepts, and components that make up the architecture. Despite all the details that we'll cover in the following sections, it is important to keep in mind that this system is fundamentally like any other artificial intelligent agent: it senses the environment through perceptors, decides on what to do next, and the *realizes* the actions, causing changes to the environment. This is, in some sense, the main loop of the agent. The major simplification in this view of the system is that it rather belittles the role of schemas, as each schema is in fact, a self-contained, independently developed, loop of Sense-Think-Act, and it is only because of the arbitrator that the whole system acts as a single coherent entity. In order to understand how schemas output behaviors they want to happen (their *proposals*), and how then the arbitration happens, we should first explain what are Resources in our system, how behaviors are specified, and what are the metadata that accompany these behaviors that help the arbitrator pick the "best" proposal.

3.2 Resources

The agent has resources at its disposal that can be allocated to actions that affect the environment. For example, speaking requires the *speech* resource, so in this example the resource is representing an actuator. Other resources in our sample implementation are Gaze (where the agent looks) and menu (the choices displayed to the user). Like any intelligent agent, at each moment in time, the agent needs to decide how to best use its limited resources to perform actions.

Resources were introduced to the architecture as a simple manifestation of the agent's physical limitations. However, further along in the development the system we realized that this concept can be utilized to capture abstract notions such as the current *focus* of the conversation. Switching to another topic of conversation is an example of a behavior that needs the focus resource ; however, glancing up to look at some movement in the background does need that resource. The usefulness of this stretch in usage of the concept becomes more clear when we later introduce schemas and their “competition” over gaining access to resources.

3.3 Behaviors

In this section we talk about behavior *specification*. Section 3.4 talks about how these specifications are “realized.” An intermediate, declarative representation of behaviors is desired as it allows us to change realizers more easily. By changing realizers we can move the system to another platform There is a well-known standard for behavior specification called Behavior Markup Language (BML). The idea behind BML was that by conforming to the standard, robotic systems will be independent of robotic platform - they can simply run on top of the appropriate BML realizer for any platform. However, in practice using BML can lead to unnecessary complications in code without any tangible pay off. We decided to import some ideas from BML (e.g., synchronization constraints), but use a custom in-memory representation for our behavior specification.

3.3.1 Primitive Behaviors

A primitive behavior is a an atomic on a single resource, e.g., saying a sentence, or nodding. These are the building blocks of any complicated behavior. The platform defines realizers for all primitive beahviors.

3.3.2 Compound Behaviors

Any collection of primitive behaviors can be a compound behavior. In its simplest form, it is a set of primitive behaviors. A compound behavior can also include synchronization constraints between those primitive behaviors. Each constraint is defined on two *synchronization points* on primitive behaviors. A synchronization point can be either *Start*, or *End*. There are three types of synchronization constraints: *Sync*, *Before*, and *After*.

Sync This constraint specifies that the two synchronization points should happen together. This is usually used to make sure two primitives start at the same time.

Before This constraints one synchronization point to happen before the other one. For example, we can specify the *end* of one head nod should happen before the *Start* of the next one. There can be a offset in milliseconds for this constraint. In case an offset was supplied, the realizer must put that much time between the two events.

After This constraint is very similar to *Before*, specifying that some behavior should happen after another one.

3.4 Realizers

Managing resources and keeping track of behaviors that are in effect on each resource is the job of the realizer component.

3.4.1 Primitive Realizers

For each primitive behavior, there should be a corresponding primitive realizer. For example, there is a one primitive realizer for speech, and there is one for looking at a specific location in the world. The set of primitive realizers should change in order to make the agent run on other platforms.

3.4.2 Compound Realizers

Just like primitive realizers, for each compound behavior there is a corresponding realizer. For compound behaviors with constraints, a PetriNet realizer is used that accepts synchronization constraints between primitive behaviors (see section 3.3.2). The PetriNet realizer is based on Holroyd's implementation [19].

3.5 Perceptors

Perceptors look for specific events or objects in the environment, or the agent itself in case of proprioceptive ones, and publish information about them to any component that is subscribed to them. A perceptor can be as simple as a motion detector that simply sets a bit if there is some movement in its vicinity, or it can be as complicated as an image processing algorithm that tracks a face and publishes its position relative to system's camera.

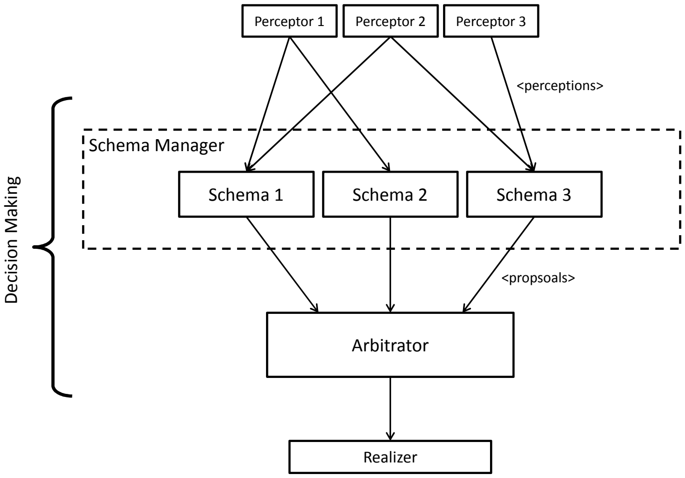


Figure 3.2: Decision making

3.6 Schemas

A schema is, in essence, a self-contained Sense-Think-Act cycle. It gets sensory information from preceptors, and decides on the best action, if any, that *contributes to its goal*. The selected action is specified as a compound behavior (see section 3.3.2). However, as there are more than one schema active at any point in time, there is no guarantee that this behavior selected by the schema will be actually performed. In other words, every schema comes up with the behavior it desires to happen, and later it'll be decided by the arbitrator component of the system which one(s) will be actually performed at any moment. All candidate behaviors, accompanied by the appropriate *parameters* are presented to the arbitrator to pick the one(s) more important to be executed now. These parameters and the arbitrator component are discussed in section 3.8.

Major benefits of this design are:

- Behavioral modules can be developed and tested independently. In behavior-based robotics, this is important because the designer of the robot/agent can add, remove, or modifiy behavioral goals independently of each other. Some of these behavioral goals are low-level, reflexive ones such as gazing at the user when idle. Others are more complicated, for example providing reminders about items on the calendar, or following a Hierarchical Task Network that represents a dialog tree (see section 5.3). In addition, this distributed architecture enables incremental addition of *levels of competence* to the agent, compatible with an iterative software design methodology.

- This modularity makes supporting content plugins for the systems straightforward. Plugins are wrapped and presented to the system as just another schema.

- In this design the concerns of designing individual schemas are (mostly) separated from the way they are going to fit together in the final system. The rules of arbitration between schemas are defined outside the schemas and can be optimized independetly. Indeed schemas need to provide some metadata about the behavior they are suggesting, including an estimate of how long the whole activity will take, how urgent it is, etc., but ideally that's all a the developer of schema needs to be concerned with, and not how and when exactly the current schema should be prefered to any other schema that may be running at the same time.

3.7 Schema Manager

The Schema Manager (figure 3.2) is in charge of starting and stopping schemas. At start up, it runs all basic and system schemas such as a simple face tracking schema, a simple movement tracking schema, schema for suggesting new activities, etc.

The schema manager also contains the plan for current interaction. In other words, this is the central activity management component that makes the system conform to the plans mentioned in section 2.2. There is a special schema, called New Activity Schema, that, on behalf of the schema manager, suggests switching to new activities that are available. Available activities are determined based the current state of the interaction plan, plus container activities at system's disposal.

3.8 Arbitrator

The arbitrator receives proposals (compound behaviors accompanied by some metadata about the activity) from schemas, and decides which ones should be executed at the current. It manages resources at the system's disposal. In a certain sense, it gives control resources to schemas, and later takes it back if another schema with a higher priority asks for them.

Chapter 4

Behavior Arbitration

At any point in time, each active behavior goal of the agent, represented as a running schema, may propose a behavior that it deems necessary. However, the agent has limited resources at its disposal; thus it must somehow pick the *best* proposed behavior(s). At each arbitration cycle, running in parallel to schema decision cycles, the arbitrator gets the latest proposal from each running schema, and between those with conflicting resources selects the best one to be *realized*.

Suppose we have a set $R = \{r_1, r_2, ..., r_n\}$ of resources, and m schemas, $s_1, s_2, ..., s_m$. When the arbitration cycle begins, the arbitrator fetches the most recent proposal p_i from schema s_i, for $i = 1, 2, ..., m$. Each proposal is a tuple $\langle a, b \rangle$, where a holds arbitration parameters (see section 4.1) and b is a behavior specification in the form of a Compound Behavior (see section 3.3.2). Let $param(p)$ denote parameters specified in the proposal p, and $behavior(p)$ denote the behavior specified in p. For a behavior b, let $res(b) \subseteq R$ denote the set of resources required. Some of the proposals may be *null*[1], implying $res(behavior(p)) = \emptyset$. If a proposal is null, the arbitrator

---

[1] It should be noted that null behaviors are different from, for instance, a behavior specifying that the agent should say *nothing*. The latter is used for what we call *idle* behaviors, e.g. when a schema has nothing to do with a resource, but at the same time does not want to release the resource to other schemas, or when the schema wants to show the user that it's thinking.

simply does not consider that proposal at all. Thus, to be absolutely precise, the arbitrator is deciding between $m' \leq m$ proposals at each cycle, but for simplicity and without loss of generality, I assume here that $m' = m$ in the rest of this chapter.

At the core of the arbitration process, we are faced with a multiple-criteria decision-making problem. For deciding which schema is most suitable for the current moment, we have multiple, sometimes conflicting criteria such as:

- the most urgent activity is preferable
- the agent should have a bias towards staying with the current activity
- in a container activity, after a while, it is good to bring up another topic of conversation
- if the current activity is going to finish soon, do not switch to another activity
- the activity most specific to the current situation is preferable

The decision is made solely based on the arbitration parameters that are part of the proposal passed down by schemas (the arbitrator does not look inside the proposed *behavior*). These parameters, introduced in detail in section 4.1, capture the urgency, remaining time, etc. of the *activity* that the proposed behavior is contributing to, and *not* the behavior itself. Remember this behavior is just the very next action that needs to be done towards the activity that the schema is pursuing.

Thus, what the function is in fact telling us is whether the agent should bother switching from a_1 to a_2. This will become more clear when the rules themselves are presented in section 4.2. We call this function $shouldSwitch$. For now, this is what we need to know about $shouldSwitch(a_1, a_2)$:

- $a1$ and $a2$ are arbitration parameters corresponding to two alternative behavior proposals
- $a1$ is bound to current activity for all the criteria that involve such notion

- the return value is a number in the interval $[0, 1]$
    - **0** meaning staying with a_1 is strongly preferred
    - **1** meaning it is strongly preferred to switch to a_2

The algorithm is presented in full on page 29. Below we discuss some details of the algorithm.

Using the *shouldSwitch* function, the outer of loop of arbitration procedure picks one of the behaviors (f) in the conflict set, compares all the others ($X = \{x_1, x_2, \ldots, x_n\}$) to that using *shouldSwitch*, finding the one with highest preference of switching. That is to say: $y = \underset{x \in X}{\operatorname{argmax}}\ shouldSwitch(param(f), param(x))$. Now, if $shouldSwitch(f, y)$ is above a threshold (currently set to 0.5) it will pick y as the preferred, otherwise f is declared the preferred one. Now let's address the question of how f should be selected. As discussed previously, there are some rules that refer to the first variable passed to *shouldSwitch* as the "current activity". For this reason, in the algorithm of page 29, the current focus of conversation is used as the first parameter to *shouldSwitch* whenever the decision involves the special *focus* resource. When deciding between lower level behaviors, the algorithm simple picks the one with higher specificity.

Through a simple example, we demonstrate that this procedure should happen iteratively to make sure that it has allocated all resources. Suppose we have three resources r_1, r_2, and r_3, and three proposals p_1, p_2, and p_3. The resources these proposals ask for are: $res(behavior(p_1)) = \{r_1, r_2\}$, $res(behavior(p_2)) = \{r_2, r_3\}$, and $res(behavior(p_3)) = \{r_3\}$. These three proposals are in the same conflict set and should be considered together. Let us suppose that after running the whole procedure mentioned above, p_1 is selected as the preferred one among the three. Now r_1 and r_2 are already allocated, but r_3 remains free. This means that even though p_3 lost the first round of arbitration, it is still possible to realize it. Thus, we should run the procedure again, this time only considering p_3, which trivially results in p_3 being selected. At this point the arbitration is done, and p_1 and p_3 are to be realized.

4.1 Arbitration Parameters

In addition to a behavior specification, proposals produced by schemas contain 7 parameters. These parameters, together with the resources needed by the behavior, are used by the arbitrator to decide if the behavior should be executed at this cycle. Below is a brief description of what these parameters are. Note that these parameters are about the activity that the behavior contributes to, and not just the behavior that is proposed.

Specificity This is the primary way to differentiate between reflexive low-level activities such as gazing at the user when there is nothing else to do with the gaze resource, and higher-level activities that are contributing to a significant goal, e.g., a conversation that is underway. The range of values for this parameter is $[0, 1]$.

Due Time This parameter captures the urgency of the activity. If there is hard time limit on when we should attend to this activity, the number of minutes till that dead-line is specified in this parameter. Thus, the range of values for this parameter is any real number bigger than or equal to 0. If there is no urgency, this parameter can be set to any arbitrary large number, depending on the arbitration rules.

Time Remaining This parameters gives an estimate of how much time, in minutes, is required for the activity to finish. For example, the rummy game plugin

Algorithm 1 Arbitration - outer decision procedure

```
Input: $p_1, p_2, ..., p_n$                                  ▷ the proposals
Output: list of Compound Behaviors

  freeResources ← allResources()
  $P \leftarrow \{p_1, p_2, ..., p_n\}$
  selected = $empty$
  while notEmpty(freeResources) and notEmpty($P$) do
      f ← $null$
      if freeResources contains focus then
          f ← previousOwnerOf(focus)                          ▷ may return null
          if f $\notin P$ then
              f ← $null$
          end if
      end if

      if f = $null$ then
          f ← $\operatorname{argmax}_{x \in P}$ specificity($param(x)$)
      end if

      $rest \leftarrow P - \{\text{f}\}$
      if $rest = \emptyset$ then
          $a$ ← f
      else
          $y \leftarrow \operatorname{argmax}_{x \in rest}$ shouldSwitch($param$(f), $param(x)$)

          if shouldSwitch($param$(f), $param(y)$) $> 0.5$ then
              $a \leftarrow y$
          else
              a ← f
          end if
      end if

      $selected \leftarrow selected \cup \{a\}$
      $freeResources$ ← freeResoures − $res(behavior(a))$

      for all x $\in P$ do
          if ($res(behavior(x))$ − freeResources) $\neq \emptyset$ then
              $P \leftarrow P - \{x\}$
          end if
      end for
  end while
```

provides this estimates based on the number of cards in players' hands, and a dialog-based schema based can calculate an estimate based on the average depth that remains in the dialog tree.

New Activity This is a boolean flag indicating whether the behavior proposed is the start of a new activity, or a continuation of an activity that we are already engaged in. As an example, all proposals coming out of the New Activity Schema have this flag set.

Container This is a boolean flag that indicates whether the activity is a container activity or not.

Good Interrupt Moment When this boolean flag is set to false, it means that this particular moment is not very good for interrupting the current activity.

Time Active This parameter is the amount of time, in minutes, that the schema behind the proposal has been active in holding at least one resource. This is the only parameter that is not set by the schema itself, the active time is tracked by the arbitrator itself.

4.2 Fuzzy Arbitration Rules

In this section, we delve into the implementation of *shouldSwitch* function, introduced at the beginning of this chapter. This function, given two sets of values for arbitration parameters gives a measure of how desirable (or undesirable) it is to switch from the first activity (i.e. first set of parameters), to the second one. A number of different criteria need to be considered in the design of this function. These criteria range from a simple "the most urgent activity is preferable," to more complicated rules involving the interplay of a container activity and other available activities. These criteria are most easily understood in terms of qualitative statements, the examples of which we have seen before. Fuzzy rule sets are a natural way of expressing these and have been used in many decision systems, for example [31]). For every parameter a number of useful fuzzy set membership functions are defined and then used in IF-THEN rules that capture the criteria. These membership functions assign a membership *grade* between 0 and 1 for variable values. For example figure 4.1 shows the three membership functions defined for the $DueIn$ parameter: *immediate*, *soon*, and *distant*. Consider the case of $DueIn = 3.0$. Its grade of membership in *immediate*, *soon*, and *distant* sets are 0.5, 0.45, and 0.0 respectively. Similar set membership functions are defined for other parameters also (*short* and *long* for *TimeActive*, *high* and *low* for *specificity*, etc.). Appendix A.1 includes the complete definitions of all of the membership functions.

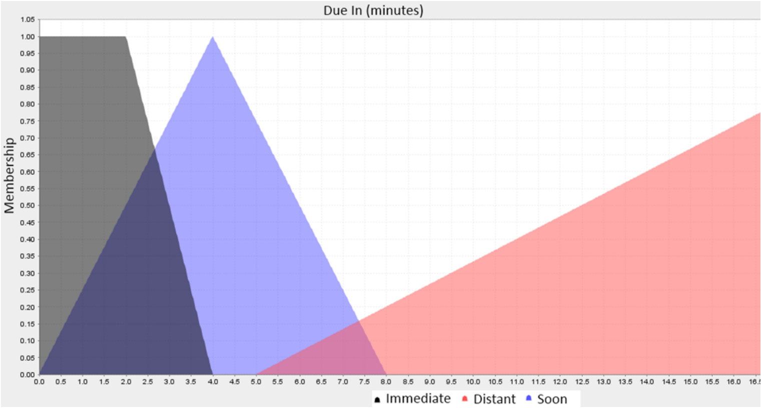


Figure 4.1: Fuzzy set memberships for *DueIn* parameter

Fuzzy rule-sets (and controllers) need to got through three steps before returning an actual result: *fuzzification*, *rule evaluation*, and *Defuzzification*. Fuzzification step is when membership functions are used against current values for all parameters. In rule evaluation, fuzzy rules are applied. Finally, defuzzification is done on the output variable, converting its current representation in terms of set membership grades to a crisp value. For a rigorous treatment of how a fuzzy rule-based system works, see [15].

The simplest rule is a conditional statements of the form `IF x is A THEN y is B`, where A and B are fuzzy sets. More complicated premises can be constructed by combining expressions using AND, OR, and NOT operators. In our system, all consequences are about the switch variable, specifically they are either `switch IS should` or `switch IS shouldNot`. With *should* and *shouldNot* being fuzzy sets on the variable switch. Each rule is evaluated and based on the activation of its premise, activates the consequent to some extent. After all rules are evaluated, these activation numbers are then accumulated, giving a us an activation value for the output variable *switch* (returned as the result of the *shouldSwitch* function).

Let us now present some of the rules. The entire rule-set is reproduced in full in appendix A.2. There are two sets of parameters, one for the first argument (the one considered the focus in the rule-set), and one for the other activity. Thus a rule with a consequent of `switch IS should`, when active, causes the whole rule-set to incline towards selecting the first activity. In reading these rules it helps to imagine the first parameter as the current activity or topic of conversation, and the second one as another activity that we consider switching to. This is not strictly true, since the algorithm in page 29 relies on these rules for comparing any two proposals, even simple ones that do not concern the focus of attention. Those that belong to the first argument are prefixed with `f_` and those belonging to the second set are have `o_` prefixed to them. We use the jFuzzyLogic[2] library in our implementation. These are represented here in the domain-specific language that jFuzzyLogic accepts the rules in. (variable names abbreviated for brevity: `due` refers to *DueIn*, `spec` to *Specificity*, and `time` to *RemainingTime*.)

```
IF f_due IS immediate OR f_time IS instant
```

---

[2] http://jfuzzylogic.sourceforge.net

```
THEN switch IS shouldNot;
```

```
IF o_due IS immediate OR o_time IS instant
THEN switch IS should;
```

The first rule above is stating that if the first activity is urgent (needs to be done in the next couple of minutes), or is just about to finish, stay with it. The second rule, says the same about the second activity. Without delving into the details of how the activation accumulation and deffuzification works, you can imagine that if, for instance, both of them are similarly urgent, these two rules will "cancel" each other.

```
IF f_time IS short AND f_spec IS NOT low
THEN switch IS shouldNot;
```

This rule states that if the time needed to finish the current activity is short (roughly around 3 minutes), and it is not a low specificity activity (it has more than 0.4 specificity), stay with it.

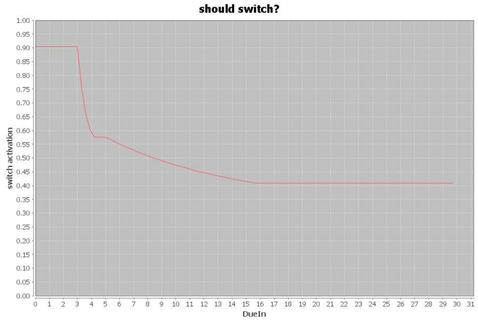
```
IF f_container Is true AND f_timeActive IS NOT veryShort
       AND o_container IS false AND o_spec IS high
THEN switch IS should;
```

This is an example of a rule about container activities. If a container activity has been active for a short while (around 3-4 minutes), it will begin to yield to other activities that are a) not containers, and b) have high specificity.

To better understand the interplay of rules and parameters, it is worthwhile to take at the effects of changing a single parameter, keeping the rest of the values fixed. Figure 4.2 show the output of *shouldSwitch* as a function of the time an activity is due. Note the fixed values of the rest of the parameters underneath the figure. It is evident that if the second activity is due in more than 8.5 minutes, the function will suggest staying with the first activity. It should not be forgotten that the value returned from *shouldSwitch* is also important in algorithm of page 29.

Figure 4.3(a) shows the effects of setting specificity to different values. Figure 4.3(b) shows the same graph, but this time the second activity is marked as *new*.

As a last example, figure 4.4 shows the effect the last rules discussed above: a container activity yielding after a while. Active time has no effect when none of the activities are containers.

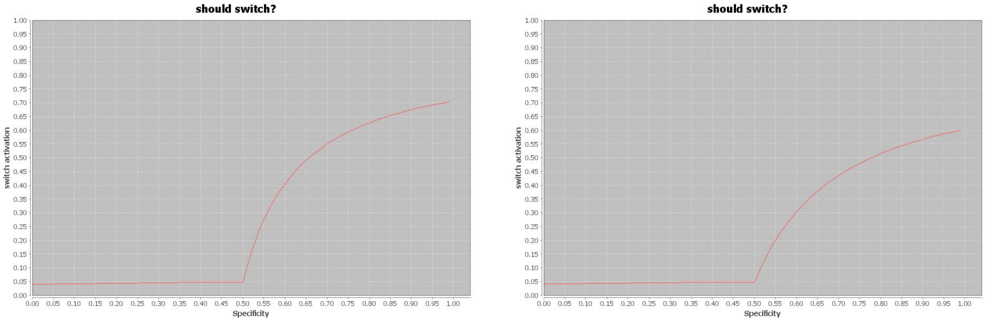


	spec	due	remaining	active	new	container	interrupt
first	0.75	50	10	3	no	no	yes
second	1	(plotted)	10	0	no	no	yes

Figure 4.2: The effect of changing the *Due In* parameter in the second activity.

4.2.1 Maintaining The Rule Set

Fuzzy logic allows us the language of domain experts in expressing the rules. Nevertheless, maintaining them is challenging work. When you change a rule, add an entirely new rule, or adjust a membership function, you can easily change the behavior of the system in ways you did not intend. A use case that previously worked will break, simply because the activation levels falls short of what was expected, or



	spec	due	remaining	active	new	container	interrupt
first	0.6	10	5	3	no	no	yes
second	(plotted)	10	5	0	*	no	yes

Figure 4.3: The effect of changing the *Specificity* parameter in the second activity, when (a) it is not a new activity, (b) it is a new activity.

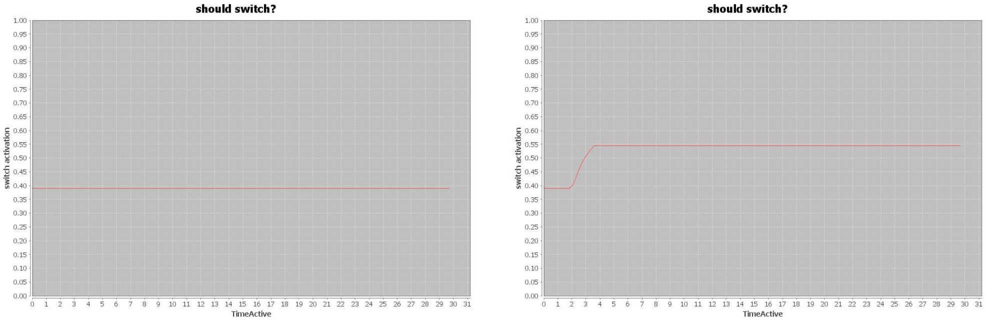
goes beyond the desired value.

We used two practical methods for making sure we understand the effects of the changes. First method is looking at graphs such as the ones in figures 4.4 and 4.2 when modifying the rules. We have developed a small utility that makes creation of these graphs fast and easy.

The second method is to adapt a methodology of test-first software development, writing acceptance tests [21]. We use an acceptance test software library called Fitnesse[3] that allows us to create readable tests in a wiki environment. Figure 4.5 shows an example of such an acceptance test. When a test is run, *reject*s and *ensure*s turn green or red, indicating success and failure respectively. This test for example makes sure that: (i) all parameters being equal, we should stay with the first activity; (ii) if the first and second have specificity of 0.75 and 0.90 respectively, we should switch to the second one; (iii) with the same difference in specificity as the previous case, if the second one is a new activity, it is not worth starting; (iv) even a new activity with specificity of 0.9 is preferred to the current one that has specificity

---

[3] http://fitnesse.org/

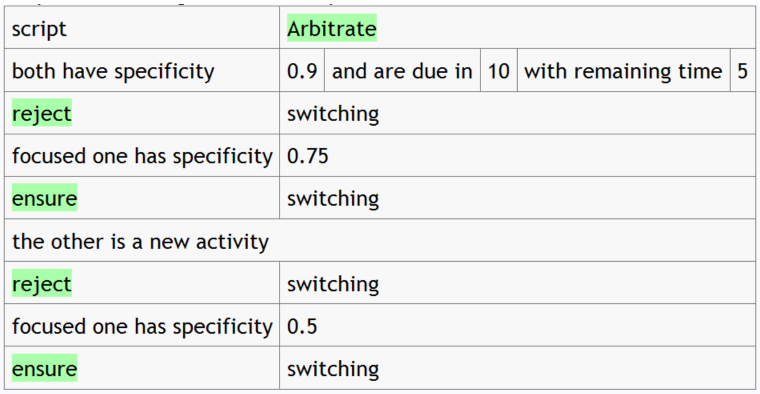


	spec	due	remaining	active	new	container	interrupt
first	0.9	10	10	(plotted)	no	*	yes
second	0.9	10	10	0	no	no	yes

Figure 4.4: The effect of changing the *TimeActive* parameter in the first activity, when (a) it is a container, (b) it is not a container.

of 0.5.

A suite of these tests exists, and with a can be run single click. Also, when a new need arises, we first add the appropriate test case(s) to the suite and then try to make them all pass again.



script	Arbitrate				
both have specificity	0.9	and are due in	10	with remaining time	5
reject	switching				
focused one has specificity	0.75				
ensure	switching				
the other is a new activity					
reject	switching				
focused one has specificity	0.5				
ensure	switching				

Figure 4.5: A Fitnesse test

Chapter 5

Additional Built-in Components

This chapter discusses some other noteworthy parts of the architecture and implementation. We begin by covering two central schemas, and two important *types* of schemas, and end our discussion with the way with which we provide higher-level abstraction for schema development.

5.1 Engagement Schema

There is a special schema in charge of greeting the user when she is about to start, or has just started to interact with the agent. This Schema is in charge of transition the system from *idle* state to *engaged*. This state of overall engagement is set in a perceptor, so that the rest of the schema can listen to it and only start actively proposing actions when we are engaged. There is nothing stopping the schemas from saying or doing something when not engaged, but it is generally a good idea to check if we are engaged with user or not.

5.2 Activity Starter Schema

With close ties with the Schema Manager component, this schema is in charge of unfolding the plan by proposing activities that are available. Behaviors proposed by this schema are either in the form proposing an activity to the user, or providing the user with a list of available activities and asking her which one she wants to do next. In either case, when an activity is selected, the relevant schema is started by Schema Manager, and it is pushed on top of the focus stack. This means that for the next arbitration cycle, this new activity will be treated as the one currently owning the focus resource (see Chapter 4).

Every behavior proposal submitted by this schema is flagged as a new activity. The rest of the parameters are currently fixed (specificity is set to a slightly lower value than that of an actual “activity” schema). While the engagement perceptor indicates that we are engaged with a user, this schema keeps proposing to start one of the available activities (if any).

5.3 Dialog-Based Schemas

As a conversational agent, it is natural to expect many dialog-based schemas in the systems. There are two specialized schema *types* that we have to accelerate dialog development: state machine based dialog schemas, and Disco-based schemas. Disco was introduced before, as a tool for creating task-based discourse plans, with a lot of advanced features for handling negotiation of goals, and generating forms of utterances automatically (i.e. without explicit authoring). For small dialog trees, or for those that involve a lot of programmatic access to user interface, databases, etc. state machine based dialog schemas are suitable as they involve shorter learning cycles for the developer of the dialog.

It should be mentioned that these types of schema receive no special treatment from the architecture – from the point of view of the arbitrator and schema manager, they are just schemas.

5.4 Plugins and Behavior Builders

In order to better support content authoring, we have created higher abstraction levels on top of schemas. This is mostly software engineering details that we will not go into the details of here. The general concept is to create some distance between content authors and the details of creating a behavior specification with all the synchronization rules.

This is achieved through the use intermediary objects we call *behavior builders*. A behavior builder is essentially a higher level way of specifying a behavior, that compiles down to *compound behaviors* before being sent to the arbitration. An example of one of these simple behavior builders is one that understands an adjacency pair, which includes an utterance and the set of response menus that need to be displayed to the user. This behavior builder has other features, such as the ability to mark a point in the utterance as the point of minimum necessary information. This mark indicates that at this point, the user probably understands what the utterance is all about, so it is safe to render the menus.

Another example is specifying a directed gaze. It is conceivable to have a behavior builder that accepts utterances decorated with references to objects in the real world. When this behavior builder is about to compile to a compound behavior, it decides about the best way to refer to the objects involved, and whether to point by fingers or not, as proposed in [19]. We do not yet have a full implementation of this behavior builder.

Chapter 6

Demonstration and Evaluation

Now let us revisit the scenario and use cases that were discussed in chapters 1 and 2 as motivations for the project. Here we are trying to demonstrate how different pieces of the design contribute to successful execution of the sample scenario. By showing how everything fits together, this chapter also serves as a complement to the previous chapters that described components individually.

6.1 Sample Scenario Revisited

This section revisits the motivating scenario of section 1.1. For each part of the interaction, a brief explanation of mechanisms and components contributing to it are provided. In addition, relevant use cases from section 2.1 are noted.

At the beginning, the agent is in idle state, meaning the engagement perceptor broadcasts the overall engagement state as *not engaged*. Face tracker schema, movement tracking schema, engagement schema, calendar reminder schema, and activity starter schema are running but not proposing anything. The first three because there is not face or movement, and the last two because the agent is not engaged in a conversation yet. Regardless, the calendar schema does not have any urgent reminders at this moment.

[Using the camera and PIR sensor, the system detects the person approaching the computer, and lights up the screen]

AGENT: Good Morning

When the user comes near the computer, her face is detected by the face perceptor. Upon receiving this perception, the face tracking schema starts proposing to track the user's face (use case: **default gazing behavior**). At the same time, the engagement schema proposes saying "Good morning," followed immediately by showing menus for the user ("Good morning", "Hi", etc.). Because there are no other schemas proposing behaviors, and these two have no resource conflicts, both get selected by the arbitrator, resulting in the agent looking at the user, saying "Good morning," and rendering the user choices on the screen. All proposals containing an utterance and then expecting a user response, are encapsulated as an Adjacency Pair by the schema, and the actual proposal containing the appropriate synchronized behavior specification is constructed by a *behavior builder* (section 5.4). The behavior specification constructed by the builder respects the minimum necessary information (MNI) marker in the utterance, and if it is not present it will render the menu two seconds after starting the speech. When a synchronized behavior such as the one just mentioned gets selected by the arbitrator, it gets realized using the PetriNet (section 3.4).

USER: Good Morning

A: Did you sleep well?

U: Yes.

A: Are you feeling. . .

[Barge-in: The person touches the menu answer before the agent is done speaking, causing the agent to stop mid-sentence]

U: Yes.

A: Glad to hear it.

The user then selects one answer. The menu perceptor broadcasts it to the rest of the system. At this point, only the engagement schema itself cares for the user response. It then continues with its built-in dialog. Whenever the user touches the menu before the speech is over, the menu perceptor receives it, the schema processes it, and moves to the next adjacency pair (or proposes silence if there is none). Either of those behaviors, causes the realizer that is currently active on the speech resource to stop, i.e. the agent stops speaking if in the middle of sentence. This is the default behavior for schemas, but the schema developer could have decided to do something else. The important point is that the system senses the user barging in, and can act upon it in a timely manner (use case: **barge-in**).

A: Do you want to know about today's weather?
U: Yes, how's today's weather?

A: It's quite warm, but it is going to rain in the afternoon.
U: Okay.

A: Do you want to hear about tomorrow too?
U: No, thanks.
⋮

When the engagement schema has exhausted all of its dialog, it sets the status of the agent to *engaged* via a system service. This change is reflected in engagement perceptor also. That change in turn causes the activity starter schema to start processing the discourse plan for the session. Suppose weather is the only activity available, aside from containers such as card game. In this case activity starter schema suggests talking about weather. This proposal is simply a utterance ("Do you want to know about today's weather?"), accompanied by two choices for the user: "yes" and "no". The user accepts this proposal, causing the activity *weather* to be pushed on top of the focus stack (i.e. becomes the current focus), and the relevant schema to be started by the schema manager. Keep in mind that the activity starter schema can accomplish all of that because it is just an extension of the schema manager component. While the weather dialog is active, the activity starter schema continues suggesting the only other available activity: the rummy game. However, the arbitration rules prevent use from starting a "new activity" that is not particularly urgent, while another one of high enough importance (the weather chat) is in focus.

A: Okay. What do you want to do?

[Menu options for calendar, rummy game, and baseball appear on the screen] U: What's on my calendar for today?

⋮

The moment the weather chat is over, the schema manager marks it as done, and the discourse plan advances forward. Suppose now we have two new options available: talking about calendar and talking about baseball. The activity starter schema proposes the agent asking "What do you want to do?", then displaying these two options, plus the option of playing a game of rummy. The user picks calendar, and an instance calendar review schema is created, and so on. Note this calendar review schema is different from the calendar *reminder* schema that was running from the start.

A: What else do you want to do?

U: Let's play some rummy.

[The rummy game is displayed on the screen]

[They start playing.]

[There are dialogs about the game that we will not reproduce here. An example is when the agent discards a card:]

A: Okay, I'm done for now. So I discard *[looks at the card]* this one. *[looks at the user]* Now it's your turn.

When calendar review is done, next available options from the plan are presented by the activity starter schema. This time the user decides to play rummy. The rummy schema is started. From time to time this schema proposes synchronized behaviors involving gaze, speech, and hand resources, the hand resource being a virtual resource for on-screen activities of the agent, in this case moving cards around (use case: **connection events – directed gaze**). For example, the agent says:

"Okay, I'm done for now. So I discard *[looks at the card]* this one. *[looks at the user]* Now it's your turn." When such a proposal is sent to the arbitrator, it causes a conflict with the face tracker schema on the gaze resource. However, it is guaranteed that the arbitrator prefers the behavior proposed by the rummy plugin over the other one, mainly because it has much higher specificity. However, the moment this behavior ends, the default gaze behavior kicks in automatically (use case: **default gazing behavior**).

> *[At some point the person leaves the computer temporarily. The agent starts an looking around, indicating that it no longer sees the user.]*
>
> *[After three minutes, the person returns.]*
>
> A: I thought you don't want to continue the game.
>
> *[Menu options appear: "No, let's continue," and "Yes, let's finish this game later"]*
>
> U: No, let's continue.

The user leaves the computer for a short while. The schema can detect this disappearance and appearance because it is monitoring the face and motion perceptors. Thus, it generates the appropriate response (use case: **appearance and disappearance of the user**). In this case the rummy schema decides to ask whether the user wants to stop the game upon user's return.

> *[A few minutes later, the agent decides to bring up another topic while they continue playing the game]*
>
> A: You know, I've been thinking. Are you interested in finding a walking buddy?
>
> ⋮

During the whole time that the rummy schema is essentially in control, the activity starter schema continues to suggest other available activities based on the current state of the discourse plan. However, it keeps loosing against the current focus, that is the rummy schema. After a while (approximately 4 minutes with the current rule-set and parameter values), and because the rummy schema sets its container parameter to true, a special fuzzy rule in the arbitrator (rule 13 in Appendix A.2) gets activated enough that causes the arbitrator to switch to an alternative activity (use case: **container activities**). In this case starting a conversation about doing exercises. This causes the focus to change to this conversation, but again after enough time another rule (rule 14 in Appendix A.2) gets activated and makes the agent to go back to the rummy game again. This is not the only way the rummy game can take back control though. It can increase its urgency and specificity when the user makes a move in the game, attempting to get back control sooner.

[The game and the conversation goes on for a few more minutes, then at 9:45]

A: Oh! It's almost ten o'clock! It's time for you to go to the lobby for the aquarium trip.

U: Okay, thanks

A: See you later.

[The rummy game is not on the screen anymore]

U: Bye

[The screens dims]

The game and the conversation continue like this for a while. In parallel, the calendar reminder schema starts proposing the agent to remind the user of an upcoming event. It sets the urgency accordingly. At some point it becomes urgent enough that the arbitrator decides to interrupt undergoing activities (use case: **event-based interruption**). The agent says "Oh! It's almost ten o'clock! ..." based on the behavior proposal of the calendar reminder schema. After this conversation, the calendar reminder schema makes a service call to terminate the current session. This puts the agent in goodbye mode. Schemas can react and say something at this moment, communicating any important information before the user leaves. The engagement schema is the one that says “See you later,” and changes the status to idle. This is the end of an interaction.

Chapter 7

Conclusion and Future Work

In this project we designed a real-time architecture that provides the essential components needed for an embodied conversation agent. Unlike previous work, here we tried to have a conversational system that supports having multiple, unrelated goals active at the same time, with a mechanism that allows the agent to switch back and forth between them. These goals are not limited to high-level conversations and activities that agent needs to attend to, but also simple reactive ones that function to make the agent interact more naturally. Examples of the latter type of goals are looking at the user's face by default, acknowledging movements in the environment, and showing appropriate verbal and non-verbal behavior when the user walks away or comes back. One key contribution is an architecture that allows all of these goals to run in parallel, each activated based on sensory information, and then one or more of them being selected for execution at any moment, based on resources they require to operate. In other words, we tried to bridge the gap between the rigid state-based discourse structure, and continuous events and necessities of the real-time interaction.

Another important contribution is that our architecture allows easy integration of content authored by different people. This is not to say that creating an entirely new behavioral schema is trivial. There are many considerations that go into the design of a new schema, such as when to activate, how to react to when the global engagement status is changed, how to react when the user has stopped responding, etc. What we have accomplished is isolating the schemas to a large degree, with clearly defined arbitration rules that can be used as the guidelines for setting arbitration parameters.

With low-level schemas and also facilities for content/schema creators, we took major steps towards decoupling *content* from *interaction* details. Generic responses to the user behavior, looking at the user, handling user barge in, and also repeating a question when necessary, are examples of interaction phenomena handled at run-time by various system components. However, there is room for creating richer abstractions that give run-time components more freedom in deciding the best turn-taking and engagement behavior. Much like the perceptor that signals the overall state of the engagement to all schemas, there could be a perceptor that observes sensory information and follows some (probabilistic) rules, such as those proposed in [6], to determine who has the floor, if the user is trying to take the floor, etc. Schemas then could use this information to adjust their behavior accordingly.

One key component of the system, which we addressed in detail in Chapter 4, is the behavior arbitration. The fuzzy rule set we have created, along with the accompanying analysis tool and suite of tests, are quite maintainable, and address the needs of our project. However, as an interesting subject of further studies, one can analyze mathematical characteristics of the rules, and compare them to other formulations that capture the same criteria. One alternative formulation is to define a partial order relation that ranks all available activities based on how desirable they are (compare with the current formulation of comparing one special option with every other option, one at a time). Two very good resources on this subject are [15] and [17].

In section 2.2 we discussed activity (topic) management in our system, the role of an activity plan created by a component outside the scope of the system. This plan is created offline, and it stays unmodified for the duration of an interaction session. Once more, it is conceivable that more of these decisions can be pushed to the run-time components of the architecture, making the agent control the flow of conversation, in words of Clark [13], in a more “opportunistic” manner. At least some utility functions can be calculated at run-time, and with the aid of a *spreading activation* model, relevant topics and activities can be initiated by the agent. Similar ideas for creating mixed-initiative systems have been suggested previously [20, 23, 12].

Bibliography

[1] James Allen, George Ferguson, and Amanda Stent. An architecture for more realistic conversational systems. In *Proceedings of the 6th international conference on Intelligent user interfaces*, IUI '01, pages 1–8, New York, NY, USA, 2001. ACM.

[2] Michael A. Arbib. Perceptual Structures and Distributed Motor Control. *Comprehensive Physiology*, 1981.

[3] R. Arkin. Motor schema based navigation for a mobile robot: An approach to programming by behavior. In *Robotics and Automation. Proceedings. 1987 IEEE International Conference on*, volume 4, pages 264–271. IEEE, March 1987.

[4] R. C. Arkin and D. MacKenzie. Temporal coordination of perceptual algorithms for mobile robot navigation. *Robotics and Automation, IEEE Transactions on*, 10(3):276–286, June 1994.

[5] Timothy W. Bickmore, Laura M. Pfeifer, and Michael K. Paasche-Orlow. Using computer agents to explain medical documents to patients with low health literacy. *Patient education and counseling*, 75(3):315–320, June 2009.

[6] Dan Bohus and Eric Horvitz. Computational Models for Multiparty Turn Taking. Technical report, Tech. Rep. Microsoft Technical Report MSR-TR-2010-115, Microsoft Research, Redmond WA, USA, 2010.

[7] Dan Bohus and Alexander I. Rudnicky. The RavenClaw dialog management framework: Architecture and systems. *Comput. Speech Lang.*, 23(3):332–361, July 2009.

[8] Michael E. Bratman, David J. Israel, and Martha E. Pollack. Plans and resource-bounded practical reasoning. *Computational intelligence*, 4(3), 1988.

[9] Timothy Brick and Matthias Scheutz. Incremental natural language processing for HRI. In *Proceedings of the ACM/IEEE international conference on Human-robot interaction*, HRI '07, pages 263–270, New York, NY, USA, 2007. ACM.

[10] Rodney A. Brooks. A Robust Layered Control System For a Mobile Robot. Technical report, Cambridge, MA, USA, 1985.

[11] Justine Cassell. *Nudge Nudge Wink Wink: Elements of Face-to-Face Conversation for Embodied Conversational Agents*. MIT Press, Cambridge, Massachusetts, first edition, 2000.

[12] Justine Cassell and Timothy Bickmore. Negotiated Collusion: Modeling Social Language and its Relationship Effects in Intelligent Agents. User Modeling and Adaptive Interfaces. 2003.

[13] Herbert H. Clark. *Using Language*. Cambridge University Press, Cambridge, 1996.

[14] William Coon. A Computational Model for Building Relationships between Humans and Virtual Agents. Master's thesis, Worcester Polytechnic Institute, 2012.

[15] J. Fodor and Marc Roubens. *Fuzzy preference modelling and multicriteria decision support*. Kluwer Academic, 1994.

[16] Michael Georgeff, Barney Pell, Martha Pollack, Milind Tambe, and Michael Wooldridge. The Belief-Desire-Intention Model of Agency Intelligent Agents V: Agents Theories, Architectures, and Languages. In Jörg P. Müller, Anand S. Rao, and Munindar P. Singh, editors, *Intelligent Agents V: Agents Theories, Architectures, and Languages*, volume 1555 of *Lecture Notes in Computer Science*, chapter 1, pages 1–10. Springer Berlin / Heidelberg, Berlin, Heidelberg, May 1999.

[17] Salvatore Greco, editor. *Multiple Criteria Decision Analysis: State of the Art Surveys*, volume 78 of *International Series in Operations Research & Management Science*. Springer-Verlag, New York, 2005.

[18] Barbara J. Grosz and Candace L. Sidner. Plans for Discourse. In Philip R. Cohen, Jerry Morgan, and Martha E. Pollack, editors, *Intentions in Communication*. MIT Press, Cambridge, MA, 1980.

[19] Aaron Holroyd, Charles Rich, Candace L. Sidner, and Brett Ponsler. Generating connection events for human-robot collaboration. In *RO-MAN, 2011 IEEE*, pages 241–246. IEEE, July 2011.

[20] Eric Horvitz. Uncertainty, Action, and Interaction: In Pursuit of Mixed-Initiative Computing. 1999.

[21] Lasse Koskela. *Test Driven: TDD and Acceptance TDD for Java Developers*. Manning, New York, 2007.

[22] Oliver Lemon, Lawrence Cavedon, and Barbara Kelly. Managing Dialogue Interaction: A Multi-Layered Approach. In *Proceedings of the 4th SIGdial Workshop on Discourse and Dialogue*, pages 168–177, 2003.

[23] Sungsoo Lim, Keunhyun Oh, and Sung-Bae Cho. A Spontaneous Topic Change of Dialogue for Conversational Agent Based on Human Cognition and Memory Agents and Artificial Intelligence. volume 129 of *Communications in Computer and Information Science*, chapter 7, pages 91–100. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[24] P. Pirjanian. Behavior coordination mechanisms-state-of-the-art. *Institute for Robotics and Intelligent Systems, School of Engineering, University of Southern California, Tech. Rep. IRIS-99-375*, 1999.

[25] A. S. Rao, M. P. Georgeff, and Others. BDI agents: From theory to practice. In *Proceedings of the first international conference on multi-agent systems (ICMAS-95)*. San Francisco, 1995.

[26] Charles Rich. Building Task-Based User Interfaces with ANSI/CEA-2018. *Computer*, 42(8):20–27, August 2009.

[27] Charles Rich, Brett Ponsler, Aaron Holroyd, and Candace L. Sidner. Recognizing engagement in human-robot interaction. In *Human-Robot Interaction (HRI), 2010 5th ACM/IEEE International Conference on*, pages 375–382. IEEE, March 2010.

[28] Charles Rich and Candace L. Sidner. COLLAGEN: A Collaboration Manager for Software Interface Agents. *User Modeling and User-Adapted Interaction*, 8(3):315–350, September 1998.

[29] Charles Rich and Candace L. Sidner. Diamondhelp: A generic collaborative task guidance system. *AI Magazine*, 28(2):33, 2007.

[30] Charles Rich, Candace L. Sidner, and Neal Lesh. Collagen: Applying Collaborative Discourse Theory to Human-Computer Interaction. *AI Magazine*, 22(4):15–25, 2001.

[31] A. Saffiotti, E.H. Ruspini, and K. Konolige. Blending reactivity and goal-directedness in a fuzzy controller. In *Fuzzy Systems, 1993., Second IEEE International Conference on*, pages 134 –139 vol.1, 1993.

[32] Sebastian Sardina and Lin Padgham. A BDI agent programming language with failure handling, declarative goals, and planning. *Autonomous Agents and Multi-Agent Systems*, 23(1):18–70, July 2011.

[33] M. Scheutz and V. Andronache. Architectural mechanisms for dynamic changes of behavior selection strategies in behavior-based systems. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 34(6):2377–2395, December 2004.

[34] K. R. Thórisson. Natural turn-taking needs no manual: Computational theory and model, from perception to action. *Multimodality in language and speech systems*, 19, 2002.

[35] Kristinn R. Thórisson. Real-time decision making in multimodal face-to-face communication. In *Proceedings of the second international conference on Autonomous agents*, AGENTS '98, pages 16–23, New York, NY, USA, 1998. ACM.

Appendix A

Fuzzy Arbitration Details

A.1 Input and Output Variables

```
VAR_INPUT
    f_due : REAL;
    o_due : REAL;

    f_time : REAL;
    o_time : REAL;

    f_spec : REAL;
    o_spec : REAL;

    f_newActivity : REAL;
    o_newActivity : REAL;

    f_container : REAL;
    o_container : REAL;

    f_goodInterruptMoment : REAL;
    o_goodInterruptMoment : REAL;

    f_timeActive : REAL;
    o_timeActive : REAL;
END_VAR

VAR_OUTPUT
    switch : REAL;
END_VAR

FUZZIFY f_due
```

```
    TERM immediate := (0, 1) (3, 1) (4, 0) ;
    TERM soon := trian 0 4 8;
    TERM distant := (5, 0) (20, 1);
END_FUZZIFY

FUZZIFY f_time
    TERM instant := (0, 1) (0.05, 0);
    TERM short := (0, 1) (2, 1) (5, 0) ;
    TERM long := (4, 0) (12, 1) (100, 1);
END_FUZZIFY

DEFUZZIFY f_spec
    TERM low := (0, 1) (0.5, 0) (1, 0);
    TERM high := (0, 0) (0.5, 0) (1, 1);
END_DEFUZZIFY

DEFUZZIFY f_newActivity
    TERM notNew := (0, 1) (0.1, 0) (1, 0);
    TERM new := (0, 0) (0.9, 0) (1, 1);
END_DEFUZZIFY

DEFUZZIFY f_container
    TERM false := (0, 1) (0.1, 0) (1, 0);
    TERM true := (0, 0) (0.9, 0) (1, 1);
END_DEFUZZIFY

DEFUZZIFY f_goodInterruptMoment
    TERM false := (0, 1) (0.1, 0) (1, 0);
    TERM true := (0, 0) (0.9, 0) (1, 1);
END_DEFUZZIFY

FUZZIFY f_timeActive
    TERM veryShort := (0, 1) (2, 1) (4, 0) ;
    TERM long := (5, 0) (12, 1) (100, 1);
END_FUZZIFY

DEFUZZIFY switch
    TERM shouldNot := (0, 1) (0.05, 1) (0.1, 0) (1, 0);
    TERM should := (0, 0) (0.9, 0) (0.95, 1) (1, 1);
    // Use 'Center Of Gravity' defuzzification method
    METHOD : COG;
    // Default value is 0 (if no rule activates defuzzifier)
    DEFAULT := 0;
```

```
END_DEFUZZIFY
```

A.2 Fuzzy Rules

```
At the beginning of the rule block below, we set the

RULEBLOCK R
    AND : MIN;
    ACT : MIN;
    ACCU : SUM;

    RULE 0: IF f_spec IS NOT low
                THEN switch IS shouldNot WITH 0.1;

    RULE 0.1: IF f_goodInterruptMoment IS false
                THEN switch IS shouldNot;

    RULE 1 : IF o_due IS immediate OR o_time IS instant
                THEN switch IS should;

    RULE 2 : IF f_due IS immediate OR f_time IS instant
                THEN switch IS shouldNot;

    RULE 3 : IF o_due IS soon AND NOT f_due IS NOT soon
                THEN switch IS should;

    RULE 3.5: IF o_due IS soon AND o_time IS short
                THEN switch IS should;

    RULE 4 : IF o_due IS distant AND o_time IS long
                THEN switch IS shouldNot;

    RULE 5: IF f_time IS short AND f_spec IS NOT low
                THEN switch IS shouldNot;

    RULE 6: IF f_due IS soon AND f_spec IS high
                THEN switch IS shouldNot;

    RULE 8: IF (o_spec IS high OR f_spec IS low) AND f_due IS NOT immediate
                THEN switch IS should;
    RULE 9: IF (f_spec IS high OR o_spec IS low) AND o_due IS NOT immediate
            THEN switch IS shouldNot;

    RULE 10: IF o_newActivity IS new AND f_spec IS high
                AND (o_due IS NOT immediate OR f_due IS immediate)
            THEN switch IS shouldNot;

    RULE 11: IF f_newActivity IS new
            THEN switch IS should WITH 0.5;

    RULE 12: IF f_spec IS low AND f_time IS instant
            THEN switch IS should;

    RULE 13: IF f_container Is true AND f_timeActive IS NOT veryShort
                AND o_container IS false AND o_spec IS high
            THEN switch IS should;

    RULE 14: IF f_container Is false AND f_timeActive IS NOT veryShort
                AND o_container IS true AND o_spec IS high
                AND o_newActivity IS NOT new
            THEN switch IS should;

END_RULEBLOCK
```