Utrecht University

Faculty of Science

Intelligent Systems Group / Division of Interaction Technology

Towards multi-modal, multi-party virtual dialog systems

by

Raoul Harel

Under supervision of

Prof. Dr. F.P.M. Dignum, Dr. Z. Yumak

Master thesis ICA-5564476

August 2017

Abstract

During the last two decades, our ability to model, render, and animate realistic-looking virtual characters has increased immensely. In contrast, progress in modes of interaction with these characters remains stagnant. For an ideally realistic method of interaction, we look towards robust modeling of conversation. However, existing methods tend to suffer from three major drawbacks: they (1) support a limited number of communication modalities, (2) are heavily geared towards one-on-one type interactions, and (3) are often built to serve in narrowly specialized scenarios. To overcome these drawbacks, we propose two tools that would aid in the development of generally applicable, multi-modal, multi-party dialog systems. The first is a system for the management of communication between actors, and the second a baseline framework for the development of dialog-related agency systems. To demonstrate their effectiveness, we combine them in a sample system inspired by social practice theory, and employ it in a virtual couples-therapy scenario as a proof-of-concept.

Foreword

This document is my master thesis, produced for the Game and Media Technology Program in Utrecht University. It documents the first steps towards the development of a generic framework for dialog systems; particularly for use in multi-modal and multi-party scenarios.

At the onset of my research, I have not already had strong convictions towards any research topic in particular. As I looked for one, I passed along many fields in computer science including graphics, animation, procedural modeling, and others. Eventually I chose to focus on the field of human-machine interaction because it is the one that most closely involves psychology — a discipline I am very much interested in.

From among the subjects in this field, I chose multi-party human-machine conversation mainly for the following reasons: (1) It is a relatively young and neglected niche, (2) it concerns a wide array of applications, and (3) it calls for creative, flexible solutions.

This work is done with the hope that it will be a solid basis for fruitful research and insights in future innovations, and I would like to thank Prof. Dr. F.P.M. Dignum and Dr. Z. Yumak for their time, guidance, advice, and support in this endeavor.

Raoul Harel, June 2017

Contents

Chapter 1

Introduction

This chapter is intended to serve as an overview of our research topic, goals, and contributions. First, we introduce the subject domain (Section 1.1), touching on human-machine conversation and briefly covering the most glaring drawbacks of current methods for human engagement with virtual characters. Next, we define our research objectives pertaining to the development of tools that facilitate the modeling of multi-modal as well as multi-party interactions (Section 1.2). We proceed with a general mention of our contributions (Section 1.3) and finally close with an outline of the document structure for the remainder of this thesis (Section 1.4).

1.1 Research domain

During the last two decades, progress in both computing software and hardware have enabled us to model, render, and animate incredibly realistic-looking virtual characters [1]. However, as these characters become ever more close to life-like in sound and appearance, progress in modes of interaction with them remains stagnant.

For example, even in the cutting edge of modern video games, dialog with non-player characters (NPC) is almost exclusively implemented as either pre-recorded sequences or as a simple mapping between textually represented choices for speech on screen and the NPC's hard-coded response [2]. More natural methods of human-machine communication can be found in personal virtual assistants, such

[1] Examples: Nvidia, *Face Works*: https://www.youtube.com/watch?v=6cZIaUEmnLI
Epic Games, *Unreal Engine*: https://www.youtube.com/watch?v=DRqMbHgBIyY
Crytek, *Cryengine*: https://www.youtube.com/watch?v=-qopI4SulZw

[2] Examples: BioWare, *Dragon Age: Inquisition*: https://www.youtube.com/watch?v=0vv1OD21Yjo
CD Projekt, *The Witcher 3: Wild Hunt*: https://www.youtube.com/watch?v=Cbm_DPQ5O-w

as Apple's Siri, Microsoft's Cortana, and Google's Now [2, 31, 36]. These personal assistants use voice recognition software to identify, execute, and respond to user intent. However, these types of interaction are almost exclusively one-way, and involve exactly two actors. In addition, one actor of the pair is extremely passive, which causes resulting dialog to constitute of a purely functional command/response pattern. These types of interaction cover only a small subset of human conversation archetypes, and even though recent efforts to remedy this are underway [38, 39], interest of the general research community in the problem is still wanting.

Naturally, the ultimate aspiration is towards the development of methods that mimic human-human communication as closely as possible, that is: robust interaction through conversation. Note that by 'conversation' we do not refer to speech exclusively. Even though it is the most obvious mode of communication, many other factors play a role as well: facial expression, cultural norms, interpersonal relationships, social roles, gaze, and others [7, 8, 11, 12, 15, 34].

Currently, existing methods for the simulation of virtual conversation scenarios tend to suffer from three major drawbacks: (1) they take into account only a limited number of communication modalities, (2) they are mostly heavily and often completely focused on one-on-one (dyadic) type of interaction, and (3) are often built to serve their purpose in a single, specialized scenario.

1.2 Research objectives

Our research focus is in finding ways to overcome the aforementioned drawbacks of current models of conversation and related dialog systems. In other words, our **primary** research objective is the following:

- Development of a robust, generic procedure for the construction of multi-modal, multi-party virtual conversation simulations.

We see the above as an objective made up of two **secondary** components relating to actors: the first dealing with their internals, and the second with their externals:

- Development of a simple generic model of an actor's internal structure and cognitive process.
- Development of an object to serve as a bridge which carries and handles communication from/to actors.

1.3 Contributions

Our **primary** contributions consist of two tools we developed as an aid for the authoring of virtual dialog scenarios:

- A system that manages data transmission between actors in conversation. It supports arbitrary quantities and types of data, and therefore naturally able to support any number of modalities. It also places no restrictions on the quantity of actors it can manage and allows for them to join or leave the conversation dynamically, thereby making it a sensible choice for simulating a multi-party setting. Finally, it is usable in both turn-based as well as real-time use-cases, and can handle synchronization of simultaneous data perception/action without ambiguity (Chapter 3).
- A baseline framework for authoring conversational agency systems. It decomposes an actor's capacity to act into several self-contained and largely-independent modules. This allows for different parties to take on and implement varying solutions to each module separately, and gives eventual system administrators the ability to freely choose and mix module implementations together in order to create an individually tailored system that is best suited for their particular needs (Chapter 4).

Our **secondary** contributions arise from the previous two:

- An agency system inspired by social practice theory. It is built on top of the framework from Chapter 4 and makes use of social expectations as triggers for both the selection as well as the timing of actions (Chapter 5).
- A demo application utilizing the social-practice-based system from Chapter 5 as a controller for virtual actors in a couples-therapy scenario. The communication management system from Chapter 3 is also being employed here. The application's main goal is to demonstrate that an agency system built with our proposed framework and running side by side with our communication management scheme is viable (Chapter 6).

1.4 Document Outline

The remainder of this thesis is structured as follows: In Chapter 2 we reflect on related literature, discuss previous work on the topic and highlight where our contributions fit in. In chapters 3 and 4, we develop two tools, respectively: a general system for managing the organization and synchronization of communications between actors, and a general framework for the authoring of conversational agency systems. Then, in Chapter 5 we use the latter to model a new approach to agency inspired by social practice theory, and combine both tools in Chapter 6 to demonstrate their operation in a sample application scenario. Finally, we end with a review of our work, the conclusions to be drawn from it, their implications, and directions for future research in Chapter 7.

Chapter 2

Background

The bulk of research into human-machine conversation is heavily skewed towards the dyadic case, making multi-party dialog a relatively young and neglected topic. However, as virtual environments become easier to manufacture, more realistic in presentation, and increasingly supportive of human-friendly input methods, the problem of modeling multi-party scenarios gains considerable appeal.

In this chapter, we briefly survey related work in virtual dialog systems to gain insight into the state of the art, including: principal archetypes of previous approaches to dialog management, related implementation frameworks/tools, and turn-taking strategies (sections 2.1 and 2.2, respectively). Finally, we summarize the major areas amenable to improvement and relate those to this work's contributions (Section 2.3).

2.1 Dialog management

Allen et al. classify five dialog management techniques according to their complexity [1]. From least- to most-complex, these are: (1) finite-state scripts, (2) frame-based approaches, (3) usage of context-sets to navigate through conversation topics, and (4) plan- and (5) agent-based models. We briefly review each in the order listed.

For dialog management in extremely simplistic use-cases, the preferred approach may be to just provide a static-script for the entire interaction. This approach is only cost-efficient for very short, linear interactions governed by a clearly defined protocol, for example: when filling in a form via an automated speech interface on the phone. Since this approach does not even scale for more complex dyadic interactions, it is of no use when modeling multi-party ones.

Only a little more complex than scripts, there is the frame-based approach. Here, the system op-

erates in a fixed context and with a single-purpose: to collect parameter values from the user so that an action depending on them may be performed. Once again, this approach covers only a small subset of possible patterns in dialog and is by definition tailored for a single user, and therefore cannot be meaningfully extended so as to have merit in multi-party scenarios.

The frame-based approach is designed to handle a single action, but what about tasks that involve several? In such cases, we can treat the entire task as if it were a set of contexts, where each context is its own frame-based action. The added complexity comes from the system now having to also detect and handle changes of the active context. Unfortunately, this extension of the frame-based approach suffers from the same issues of its predecessor when it comes to multi-party dialog, and so we must turn our attention towards more generic approaches.

The next class of methods to review is plan-based models. This approach refers to a joint effort of both the system and its user in composition of a procedure — the plan — to satisfy some task. This approach is still most commonly used for dyadic interactions between one system instance and one user, but there is no theoretical barrier into extending it to support more than one. However, even with multiple users interaction is still pair-wise, that is to say: bidirectional between each user and the system, but not including users communicating with each other. We are looking for something more general than that.

Agent-based models are the most complex of the hierarchy, and are similar to the plan-based approach except that they aim to deal with tasks that take place in a dynamic environment. Notice that all methods listed (and the vast majority of work done in dialog management in general) pertains to a narrow set of interactions, namely: task-oriented ones [19]. This is understandable; firstly, the need for task-centric dialog systems dominates most practical applications. And secondly, the structured, procedural nature of the tasks themselves is mirrored in the dialog patterns surrounding them, making these relatively easier to model than other types of interaction. That being said, we are specifically interested in modeling human-human conversational patterns, i.e. where the system plays the role of another actor and not that of a crutch used to achieve some technical goal. Potential applications for systems capable of that include: the entertainment industry (video games), education (educational games), and training of professionals (serious games).

Admittedly, as far as we can tell there is little previous work done in this area. However, there is one particular approach to dialog modeling that we would like to mention: Trindikit. As part of the TRINDI project, Larsson et al. introduce a toolkit to aid in the development of information-state-based dialog systems, named Trindikit [16, 17, 20]. The information state approach assigns each system its own state

object, which is then updated in response to changes in the environment and actions of other actors in the scene. Modeling the relationship between these external events and changes to the information state are so-called update rules. Each rule is made up of a precondition and an effect — to be applied onto the state when a rule is activated. However, Trindikit only supplies this formulation as a basis for the dialog system, and it is rather the users themselves who are left to decide on the information state structure, the update rules present, and the control algorithm managing their activation.

We think that the information state approach used by Trindikit is definitely a step in the right direction, but we are concerned that dialog management using only update rules alone would not scale to systems of higher complexity. To address this, we think there is a place for an additional layer of control on top of the original approach: one that (1) captures all domain-agnostic aspects of multi-party dialog and (2) allows for more intuitive abstractions for the arrangement and execution of update rules. We address the former in Chapter 4, where we outline a framework (borrowing from Trindikit) designed specifically with multi-party scenarios in mind; and the latter in Chapter 5, where we present a social-practice-based approach that handles update rules via a more accessible abstraction: social expectations.

2.2 Turn-taking

In this section we briefly survey the principal strategies employed to model the turn-taking aspect of dialog. These methods are divided among three categories: passive-wait, decision-theoretic, and data-driven (sections 2.2.1–3, respectively). Following the survey, we identify and discuss a common shortcoming of the methods reviewed (Section 2.2.4) — one which we address in later chapters.

2.2.1 Passive-wait strategies

A recurring theme in turn-taking models is the desire to avoid overlapping speech. Pragmatically, this makes sense: As it is harder to process, tandem speech undermines one of the principal goals of conversation — the dissemination of information. Empirical observation confirms this tendency: although varying to some extents depending on culture, overlapping speech is universally the exception rather than the rule [34]. Therefore, a straightforward strategy for overlap avoidance in conversation is to never interrupt and wait until everyone else has concluded their turn before taking ours. The challenge here is recognizing when that is the case, i.e. when does a turn end?

Pauses in speech alone are not a reliable indicator for turn-endings, as they also naturally occur between consecutive utterances or as part of intonation. To that end, passive-wait methods rely on

various end-of-turn detection techniques that take into account signals from multiple modalities such as prosody, lexical indicators, eye-gaze, gestures, and others [7, 8, 12, 15, 21, 37].

Passive-wait methods excel in scenarios where the conversation is largely one-sided: from human to machine. However, in other settings the reluctance to interrupt gives out a cold, unnatural vibe to the artificial party. This because although overlapping speech in human conversation is limited, it is not without its purpose and sometimes it is even desired. For example: when interrupting the current speaker to address a misunderstanding, or when counting down together towards the start of a new year. Therefore, passive-wait methods are too limited to be generally applicable to a dynamic multi-party setting.

2.2.2 Decision-theoretic strategies

If we view actors as beings in possession of goals, and their engagement in conversation as a means to meet these goals, then we may transform the problem of turn-taking into a decision-theoretic one. In a decision-theoretic formulation, agents have — during specific windows in time — the option to either bid for the dialog floor themselves or surrender it to others. Prevailing methods to make that decision involve either following a static rule-based recipe or consulting a utility cost-function.

Decisions through static rules

The influential model developed by Sacks, Schegloff, and Jefferson (SSJ) is an example of a rule-based solution to the bid-or-not decision problem. Although it is one of the oldest models and despite being derived from dyadic conversation, it is still widely used as a basis for new turn-taking methods [13, 33]. It can be summarized in the following triplet of guidelines [30]:

1. The last actor to speak may explicitly allocate the next turn to another party.
2. If the current speaker does not select a successor, anyone may bid and acquire the floor.
3. If none of the other participants bids, the last actor to speak may take an additional turn.

This simple answer to the turn-taking problem has its flaws [25]. Firstly, it assumes turns are discrete, and that interruptions or overlapping speech are non-existent: As seen in [13], an attempt is made to combine the SSJ model with cultural parameters and it is found that the observed degree of overlapping speech in Spanish does not agree with the discrete-turns assumption. Secondly, the SSJ model (and derivatives) does not extend to multi-party settings well as it does nothing to address what should be done in a state of simultaneous bids by multiple actors. Thirdly, it assumes perfect content

transmission which is not guaranteed in practice: e.g. when the last speaker allocates the next turn to another actor, a misunderstanding might cause a different actor to wrongfully assume it has the floor.

Utility-based decision costs

Another approach to the turn-taking decision problem is the assignment of a numerical value to each choice representing its utility cost. In this context, 'utility' often refers to a quantified contribution of a choice — either bid or surrender — to the actor's goals. The simplest cost-functions are nothing but simple heuristics, often tailored for specific conversational scenarios [3, 4, 32]. But, more complex functions may incorporate the mental state of an actor, its beliefs about other actors, and even a capability to predict future effects of a given choice on the conversation's state [16, 26, 28].

Utility-based approaches are attractive due to their flexible nature. However, their effectiveness depends entirely on a suitable formulation of the cost-function, which is far from a trivial requirement to satisfy. Naturally, it is in our interest to choose a function that resembles the internal decision making process humans use as closely as possible, however — as is often the case in such matters — a boost to realism comes with the price of additional complexity.

2.2.3 Data-driven behavior synthesis

Data-driven methods are a popular choice for the simulation of commonly occurring patterns in conversation. They make use of available corpora to infer relevant distribution parameters of important conversational features. In [18] the ICSI meeting corpus is used to train a probabilistic model capable of generating speech/non-speech patterns for multi-party conversation [14], but is not put to the test in a real-time environment.

Such methods are useful when it is desired to synthesize mock-conversations between virtual characters alone, but are unsuitable for use in interactions involving humans for two reasons: First, the resulting model is dependent on the training-set; and second, it is not flexible enough to handle edge cases. Dependence on the training-set is crippling, since most datasets we have at our disposable are somewhat lacking: They tend to focus on only a narrow set of scenario types, and are often insufficiently annotated [6]. Lack of flexibility is another drawback that only manifests itself when humans are involved. That is to say: although the model will behave adequately on average, whenever an exceptional event occurs it will become glaringly obvious to any human observer that the model does not produce realistic behavior.

2.2.4 Discussion

A common shortcoming of the methods reviewed is that they all operate under the implicit assumption that a turn of dialog has instantaneous duration. To be fair, this assumption is warranted in those applications where the nature of the conversation is either one-sided (human orders/queries machine) or in those that obey an exceedingly strict protocol. As was already mentioned in the introductory chapter, so far most dialog systems were made to operate in exactly these kinds of conditions, and so assuming turns are discrete and without duration did not cause much trouble.

However, if we wish virtual dialog to play a greater role in more complex, dynamic applications such as those often found in serious-games or real-time discourse of a less-than-formal nature, then this issue must be addressed. If it is not, then it follows that once an actor takes a turn, it proceeds to complete it with no regard to any new events that transpire during it. In other words: Assuming instantaneous turns necessarily implies a refusal to handle interruption — a big deal in multi-party scenarios, considering that the more actors participate, the more likely it is interruption will materialize. In Chapter 4 we present an agency framework for conversational actors that rectifies this and treats turns as having duration, and in Chapter 5 we use it to propose one approach towards the handling of interruption.

2.3 Summary

In this chapter we review previous related work regarding both dialog management as well as the turn-taking aspect of virtual conversations. In dialog management, we identify that current research pertains to only a limited subset of all dialog interactions, concentrating on task-oriented scenarios whereas we are more interested in the general case. In turn-taking, we highlight how existing methods neglect to properly handle interruption. And lastly, in both fields we detect a lack of research interest in multi-party dialog. Our contribution to mending each of these problems is spread over the remainder of the thesis: In chapters 3 and 4 we develop dialog-modeling tools designed specifically for dealing with multi-party scenarios, and in Chapter 5 a new information-state-based approach to scalable dialog management and turn-taking — including the handling of interruption.

Chapter 3

Communication management system

In this chapter, we outline our formulation of a general system that manages communication between actors in multi-modal, multi-party conversation scenarios. We begin with a general overview of the perspective through which we view conversation itself (Section 3.1), and then proceed to define the problem domain alongside relevant research questions to answer (Section 3.2). Beside those, we list several additional requirements that we look for a solution to satisfy in order for it to be usable in practice (Section 3.3), and then propose one in the form of a centralized system to act as transmission-coordinator between actors (Section 3.4). Finally, we close the chapter with a discussion of the system's customization options and a summary (sections 3.5 and 3.6, respectively).

3.1 Domain overview

We view conversation as a continuous, dynamic process between two or more actors. For as long as they are engaged in dialog, we attribute each individual actor's agency to a rudimentary routine which is executed internally. On an abstract level, we describe this routine as consisting of three steps which are being iterated over in a loop, with the steps being: (1) perception, (2) deliberation, and (3) action [10, 27].

In the first step, an actor perceives external stimuli that took place since the routine's last iteration; in the second, these stimuli are processed in order to determine and apply their effects to the actor's internal state; and in the third and final step, an actor may produce its own actions in response to its new state. The resulting behavior will then be perceived by other actors, and the cycle continues on.

We term this routine 'the update routine', because its function is to keep an actor's internal state and actions accordant (up to date) with the external environment within which it is embedded and also

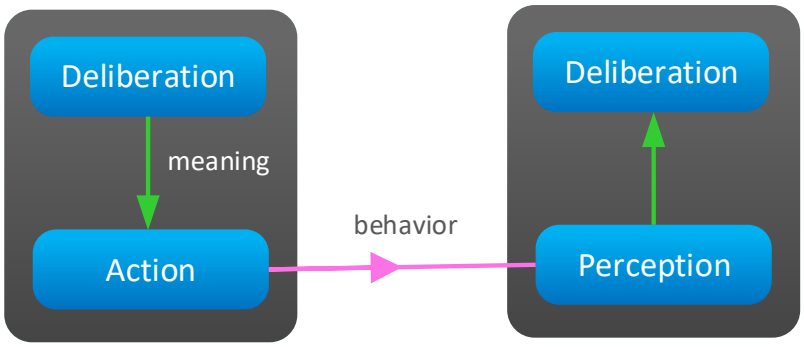


Figure 3.1: The transmission process of meanings from one actor to another in four steps: (1) The authoring of a meaning to transmit, (2) realization of that meaning through behavior, (3) perception of the behavior by the recipient, and (4) incorporation of the interpreted meaning into the recipient's cognitive process.

through time as well. So far, our description of the update routine could be applied to actors in any context. However, when modeling conversation specifically, each of the three steps can be expanded to include additional details.

For starters, knowing our context, we can define both the aforementioned external stimuli perceived as well as the actions produced by actors as one and the same: a stream of data representing behavior such as speech, hand gestures, facial expression, and other modalities of communication.

The purpose of such behavior is to carry some semantic content — a *meaning* — from one actor to another. It is the medium through which these meanings travel. Taking that into consideration encourages us to recognize that on the whole, one may consider the meaning transmission process as a twofold transformation: from meanings to behavior and back again; and that a successful transmission relies on both an adequate realization of a meaning as well as correct interpretation of the mediating behavior (Figure 3.1).

3.2 Problem definition

In this chapter, we are concerned with modeling those aspects of the meaning transmission process external to the actors themselves (for more insight into the modeling of actors internally, see Chapter 4). That is, pertaining exclusively to the organization and relay of the behavioral data across actors in a multi-party, multi-modal conversation scenario. In pursuit of similar goals, v. Oijen & Dignum (2012)

develop an approach that facilitates perception/realization of communication data by linking it with both the cognitive process of actors as well as the engine governing the simulation [22]. In our scope however, we limit ourselves to using transmitted data alone, and are looking to answer the following principal questions:

1. How should the data in transit be organized?
2. How should the data in transit be synchronized?

The first question is brought up by the fact that the data we are dealing with here is multi-modal, and therefore not homogeneous in nature. We rather expect it to be made up of various types, quantities, and frequencies.

To illustrate this point, consider a comparison between one data stream encoding speech and another relaying eye gaze information. Aside from their obvious difference in type, they also differ in quantity and frequency of occurrence: Speech data will most likely consist of lengthy strings of text, whereas eye gaze information has a relatively small footprint (e.g. a direction vector) and would occur in short bursts.

An additional indication of the need for data organization is the idea that an actor might only be interested in a subset of the data streams (read: communication modalities) present in a scene, implying that whatever method chosen to represent and store data in transit should support a straightforward filtering mechanism.

The second question is motivated by the problem of simultaneous data transmission. This refers to the case where two or more pieces of data are emitted simultaneously. Data that — upon arrival at recipient actors — must also be perceived as a simultaneous occurrence. If it were not so, then the order in which different actors process the data would remain ambiguous. This underlines the need for an external synchronization scheme.

3.3 Requirements

During development of a system that would address those challenges mentioned in the previous section, we stuck closely to a premeditated set of requirements. The following is a listing of our primary ones — conditions that when left unsatisfied will render the system untenable:

1. The system **must** support a variable number of data types.
2. The system **must** place no restrictions on the types of data allowed.
3. The system **must** support a variable number of actors.
4. The system **must** make no assumptions about the internals of actors.

The principal motivation behind these requirements is the aspiration to maximize the system's usability and minimize its adoption cost. Requirements two and four ensure that already existing projects need not overhaul major implementation details in order to conform to some externally-mandated standard. At the same time, requirements one and three promise consumers that the system is scalable even up to highly complex use-cases involving many actors.

3.4 Architecture

At the base of the system is a single object, the communication *manager*. Its job is to act as a postmaster of sorts: actors submit data they wish be broadcast to the manager, and likewise they are also notified by it to read data broadcast by others. We use the word 'notified' here because the relationship between actors and the manager follows the observer pattern. That is, where actors (observers) must first register the types of data they are interested in with the manager, and only from then on will they be granted access to communications of that type — whenever available.

3.4.1 Data synchronization

Recall from Section 3.1 that each actor repeatedly undergoes an individual tri-step update routine, with the steps being: (1) perception, (2) processing, and (3) action. The communication manager itself also contains its own update routine — a global one — that coordinates the update routines of each of the actors in the scene and handles any data they may emit. This global update routine is listed below:

Algorithm 3.1 One iteration of the global update routine.

1: $A \leftarrow$ a set of all actors in the scene
2: $D_{in} \leftarrow$ a set of all data emitted during the previous iteration
3: $D_{out} \leftarrow \{\}$
4: **for each** actor $p \in A$ **do**
5: $\quad D_p \leftarrow \{d \in D_{in} \mid d \text{ is of interest to } p\}$
6: $\quad$ let p process D_p
7: $\quad$ let p act and add any data emitted by p to D_{out}
8: **end for**

Note that a sort of double-buffering is taking place. In order to ensure that the set of data perceived by all actors (D_{in}) remains constant throughout the iteration, we write any output data into a second buffer (D_{out}). Notice that come the next iteration, D_{in} will effectively contain the value that D_{out} carries at the end of the current one, and so a buffer swap has taken place.

In addition, note that this routine takes care of the problem of simultaneous transmission mentioned in Section 3.2: actions that happen during a given iteration (i.e. simultaneously) will be presented to actors in the one following it as a set, so they may be processed as such without ambiguity.

3.4.2 Packets and channels

With the general architecture explained, let us get down to the details. Firstly, we must point out that data submitted by actors is not stored inside the communication manager as-is, but as a payload within *packets*. Packets contain — aside from the data itself — an additional piece of meta-information: the identity of that data's author. This is so in order for actors on the receiving end to be able to associate perceived data to its source, as is the case in reality.

With that being said, it is true that this piece of information could have been embedded within the data itself. After all, the data's underlying structure is supplied by the consumers of the system, and they have total freedom over it. However, seeing as it is such a vital detail, and in practice almost universally required for any dialog system to function, we believe we are justified in unburdening the user of this common task and providing a working solution out of the box.

Once a piece of data is packaged as a packet, it is ready to be stored in a *channel*. A channel is simply a container for packets, and the manager contains one per data type occurring in the scene. In accordance with the routine listed in Algorithm 3.1, channels are double-buffered. That is to say: all accessor operations act on a front buffer while all mutator operations act on a hidden back buffer.

Taking packets and channels into account, a more detailed version of the global update routine can be described as follows:

Algorithm 3.2 Detailed description of one iteration of the global update routine.

1: $A \leftarrow$ a set of all actors in the scene
2: $C \leftarrow$ a set of all data channels
3: **for each** actor $p \in A$ **do**
4: $\quad C_p \leftarrow \{c \in C \mid c$ carries data of a type that is of interest to $p\}$
5: $\quad$ let p process all data on the front buffers of C_p
6: $\quad$ let p act and add any data emitted by p to the back buffers of C
7: **end for**
8: swap the front and back buffers in C
9: clear all back buffers in C

A graphical illustration of this procedure is also visible in Figure 3.2, with its description available below:

(a) The actor on the left processes the packets on the front buffer.
(b) The actor on the left emits new data to the back buffer.
(c) The actor on the right processes the packets on the front buffer.
(d) The actor on the right emits new data to the back buffer.
(e) The front and back buffers are swapped.
(f) The new (formerly front) back buffer is cleared of data in anticipation of the next iteration.

3.4.3 Centralized vs. distributed design

One might ask: why opt for a centralized approach with a single object acting as middle-man? Why not use a distributed scheme, where each actor broadcasts its behavior to all others directly? Two factors contributed to this choice of design: Firstly, direct contact between actors entails that each of them must be aware of all others. This inhibits the representation of eavesdroppers — actors who are listening in on the conversation without others being aware of them. Secondly, a distributed approach introduces data synchronization issues: e.g. there is no straightforward way to indicate simultaneous data broadcasts because in such a distributed scheme a set of simultaneously occurring actions would be mistakenly perceived as a sequence (of arbitrary order, no less). In contrast, the communication

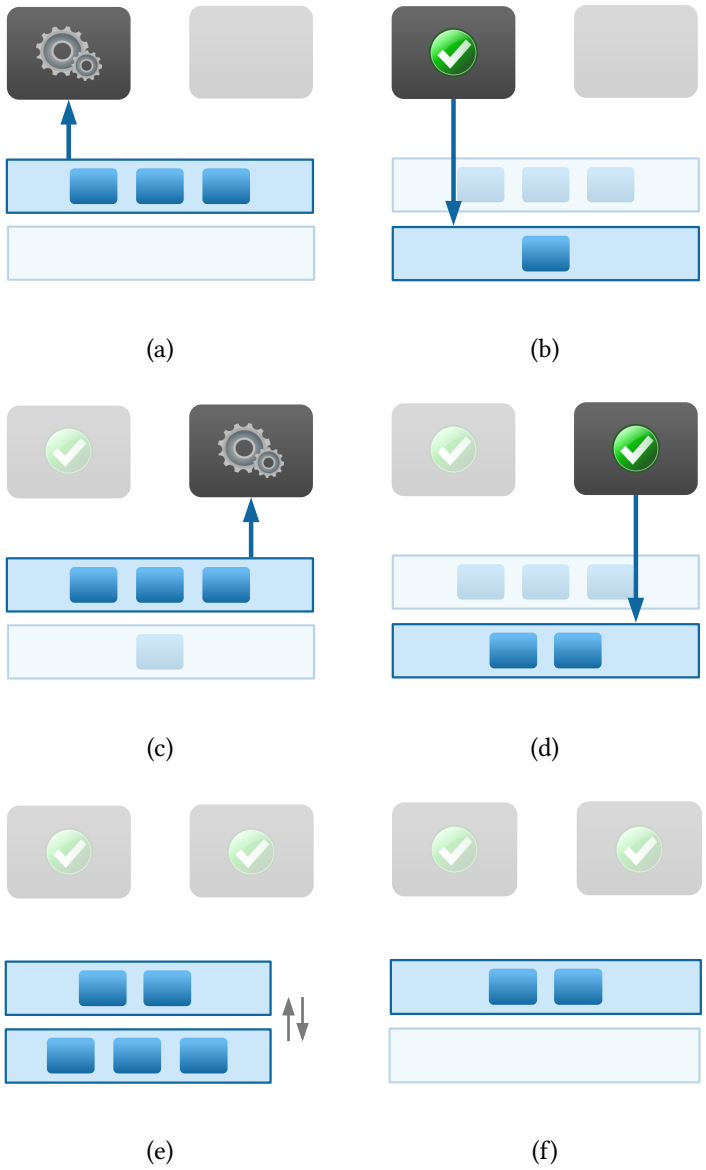


Figure 3.2: An illustration of one iteration of the communication manager's global update routine. Depicted are two actors (gray), a manager containing one data channel (light blue) with both the front and back buffers visible (upper and lower blocks, respectively). Data packets are also visible within the buffers (dark blue).

manager formulation allows us to buffer incoming data and then present it as a simultaneous occurrence when queried.

3.5 Usage and customization

In the interest of generality and reduction of adoption cost, the communication management scheme we outline contains three angles through which administrators of a virtual conversation scenario can customize its behavior to suit their specific use-case. Those are: (1) specification of supported data types, (2) actor engagement and disengagement, and (3) update routine call scheduling. In the remainder of this section we expand on each of these in order.

3.5.1 Specification of supported data types

The first step needed to be taken when initializing a new instance of the system, is the specification of the data types to support. That is, a declaration of all types we expect to be submitted to the communication manager during operation. We consider this to be a static piece of information that is known beforehand by the administrators and which remains constant throughout the scenario. In case of an actor attempting to submit data of an unsupported type, the data is discarded. While this feature is redundant in those cases where all actors are supplied by the scenario's administrator, it does help in enforcing of expectations when actors are provided by a third-party.

3.5.2 Actor engagement

Once the system is initialized and its supported data types specified, actors who wish to be included in the global update routine (mentioned in Section 3.4.1) must register with the communication manager both their identity as well as the set of data types they are interested in perceiving. Actors are free to disengage from the conversation or to change their data subscription preferences at any time.

3.5.3 Scheduling of the update routine

An initialized system serving some actors is ready to begin execution of the global update routine. Here, the scheduling of calls to that routine is completely up to the administrator and dependent on the use-case, e.g. real-time applications may invoke the routine many times per second, while turn-based scenarios — such as those often found in serious games — would initiate an update in response to user demand.

3.6 Summary

In this chapter we outline a formulation of a communication management system between actors, intended for use in virtual conversation modeling. It supports multi-modal as well as multi-party scenarios, synchronization of simultaneous perception/action, dynamic actor (dis)engagement, and is capable of being customized to suit a wide range of use-cases. In Chapter 4, we introduce a framework to facilitate development of agency systems to model actors themselves, and in Chapter 6 we combine both tools in a prototype application as a proof of concept.

Chapter 4

Framework for agency systems

In this chapter, we present a general framework for the modeling of agency in conversational actors. We open with a description of the framework's domain, and map out an abstraction of the mental process governing an actor's behavior (Section 4.1). Next, we illustrate the need for a baseline framework embodying that process and list its principal goals (Section 4.2) and requirements (Section 4.3). We then proceed to present our proposed architecture for such a framework (section 4.4), demonstrate how it is to be used (Section 4.5), and end with a summary of this chapter's material (Section 4.6).

4.1 Domain overview

In this section we gradually map out an abstraction of what we call 'the agency process' — the process that governs an actor's entire capacity to act. That is, everything beginning at the perception of input and up to and including the production of output. We begin with an overview of an actor as regarded from the outside (Section 4.1.1), then segue into a brief discussion about actor behavior as carrier of meanings (Section 4.1.2), and finally describe in detail the three principal stages that make up the agency process itself (sections 4.1.3–5).

4.1.1 Actors and agency

We view actors as stateful entities hosting a capacity for cognition within them. Their cognitive process takes on input from the external environment they are embedded in, and pairs it with their current state in order to produce a new state that reflects absorption of that new information. Then, as a final step the process produces action based on that new state as output to the environment.

If we were to condense this description into its bare essentials, we would say it consists of three

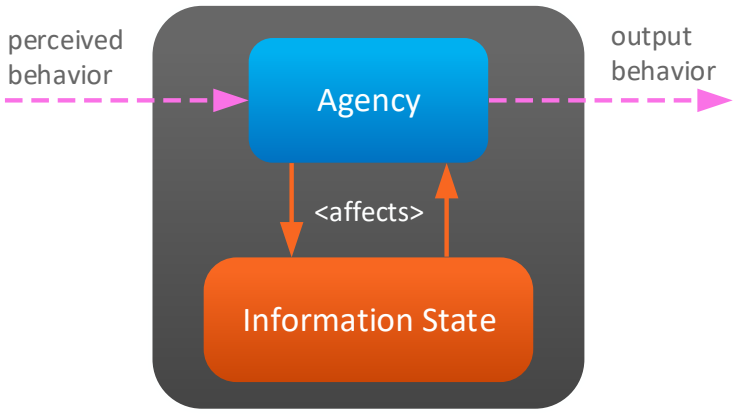


Figure 4.1: The relation between an actor, its cognitive process, its state, and the environment.

stages: (1) perception, (2) deliberation, and (3) action [10, 27]. When discussing conversational actors specifically, a description of these components may be understood to mean, respectively: (1) the perception of other actors' behavior, (2) an update of the actor's own internal state, leading to selection of an action as response, and (3) the timing and realization of said response — again, through behavior. We call this routine 'the agency process', because it is both a sufficient and necessary account for an intelligent actor's capacity to act (Figure 4.1). For discussion about our proposed mechanism for bridging the gap between one actor's action and its perception by another, see Chapter 3. This chapter however is dedicated to the modeling of the agency process itself.

4.1.2 Meanings and dialog moves

There is a distinction to be made between a communicated intention and the behavior expressing it. Even though behavior is the concrete element flowing in and out of the agency process on both ends, it is not the element we wish to reason about internally within it. Instead, we much rather reason about the semantic content behavior carries: *meanings* [22].

The reason for our added interest in meanings is that any discussion about behavior itself will always be more context-dependent, more restricted in scope, and more limited in application. Put another way: meaning is a constant, and behavior is its variable manifestation whose exact form depends on circumstance.

Therefore, in order to minimize such a dependence and as we strive to have any further discussion

be applicable to as wide a set of circumstances as possible, we shall shift our focus almost exclusively to the role of meanings rather than behavior from now on.

In order to facilitate further discussion about meanings and their role, we label the act of communicating a meaning through behavior as the realization of a so-called *dialog move*. We view actors as producers of a constant stream of dialog moves — from the moment they enter a conversation and up to the moment of exit. Furthermore, since even "idle" behavior has a communicative quality to it and therefore must be regarded as a move in and of itself, we regard such a stream of moves to be without gaps. This means that it is impossible for an actor to *not* be realizing any moves for as long as it is engaged in dialog.

Another observation we wish to make refers to the existence of an important universal move: the *idle* move. It is readily apparent that this is the move we naturally default to. Therefore, even though the set of moves available for an actor may be heavily dependent upon scenario specificities, it is guaranteed that the idle move would always have its unique role to play in any setting.

Armed with dialog moves as an umbrella term to denote all actions perceived/produced by actors, we are now in a position to incorporate them in our description of the agency process, which now would read as: (1) the interpretation of perceived behavior as the realization of dialog moves, (2) an update of the state and selection of a target move to make in response, and (3) timing and realization of that target move through behavior (Figure 4.2).

We elaborate on each of the main three stages that make up the agency process — perception, deliberation, and action — in that order and within the next three sections 4.1.3–5.

4.1.3 Perception stage of agency

Perceiving the behavior of other actors is a task we consider to be comprised of two temporally-related components: (1) perception of past recent activity up to but not including the present, and (2) perception of current activity.

In terms of dialog moves, recent activity perception translates to perceiving those moves that have taken place in the span of time beginning at the conclusion of the latest previous iteration of the agency process and ending at the onset of its current one.

As for current activity: even though we would have liked to be able to know what moves are in the process of being realized at present, we cannot. This is so because the act of realizing a dialog move is not an instantaneous event, but rather an eventual one. This entails an inconvenient consequence: we are not allowed to assume we know what dialog move is taking place until its realization is complete.

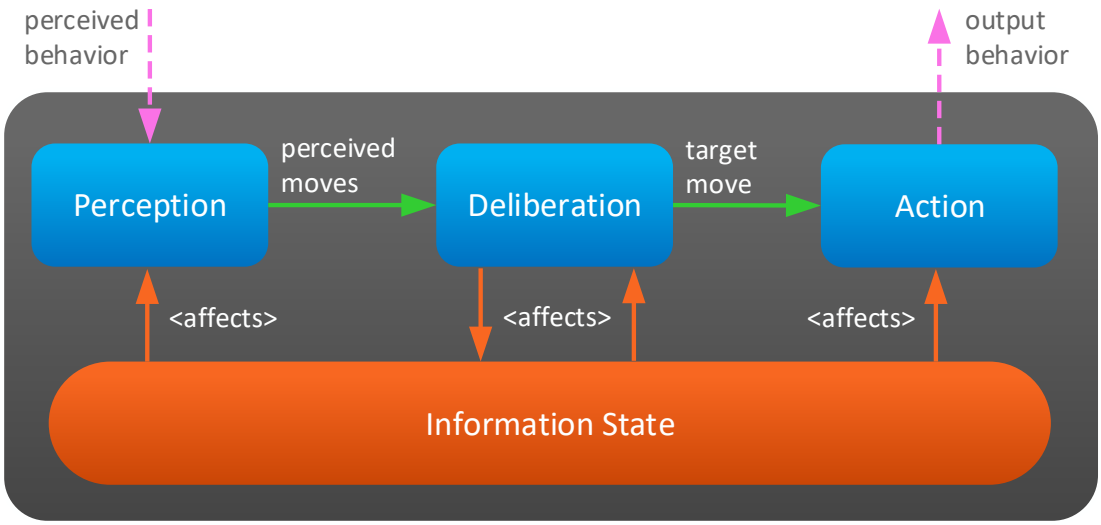


Figure 4.2: General description of the agency process. Emphasized is the contrast between interactions with the external dealing in behavior (dashed pink), and the internal dealing in meanings (solid green).

However, there is at least one important exception to this rule: the idle move.

Unlike most moves, we consider the idle move (Section 4.1.2) to be realizable in an instant. This enables us to immediately recognize the difference between an actor realizing it and one that does not — a vital piece of information that becomes particularly useful when deciding on the timing of an actor's own action. Also note that this simplification is not without justification; we are hard-pressed to come up with any practical scenario where an actor does not possess such an ability.

And so, the perception stage of the agency process is one that accomplishes a conversion of perceived behavior into two forms of output: dialog moves representing recent activity, and a classification of actors into passive (idle) and active (non-idle) categories representing current activity (Figure 4.3). Both outputs are then fed into the deliberation stage, which is discussed in the next section.

4.1.4 Deliberation stage of agency

The deliberation stage of the agency process is where an actor translates perceived information into mutations of its state, and then uses that new state to select an action to be made.

Updating of the state is an operation that takes on three inputs. Two of which are in fact the output of the perception stage (Section 4.1.3), representing recent and current perceived activity of outside

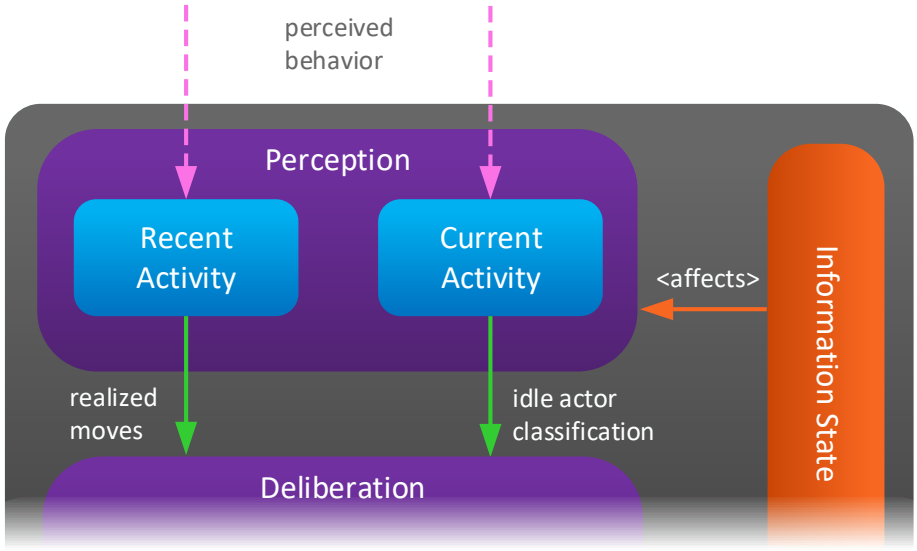


Figure 4.3: The perception stage of the agency process.

actors. The third input is implicitly known by the actor itself, as it is a snapshot of the actor's own activity as expressed by three dialog moves.

The three moves that make up an actor's own activity snapshot are termed: the *recent move*, *target move*, and *actual move*. The recent move refers to the latest move the actor has completely realized, the target move to the move the actor wishes to realize currently, and the actual move to the move the actor is actually in the process of realizing.

Proceeding the update, the first operation to make use of the new state is the one responsible for action selection. Here, the actor determines what move ought to become its new target move for the duration of the current agency process iteration.

To summarize: the deliberation stage takes into account activity both externally perceived as well as of the self, uses it to determine and perform updates to the state, and then bases its selection of a new target move on that new state (Figure 4.4). Once selected, the target move is forwarded to the action stage where it may or may not be realized. Details are in the upcoming section.

4.1.5 Action stage of agency

The action stage is where the decision is made on whether or not to realize the actor's target dialog move, and in the case where it is to be realized, this stage is also the one responsible for the move's expression through output behavior.

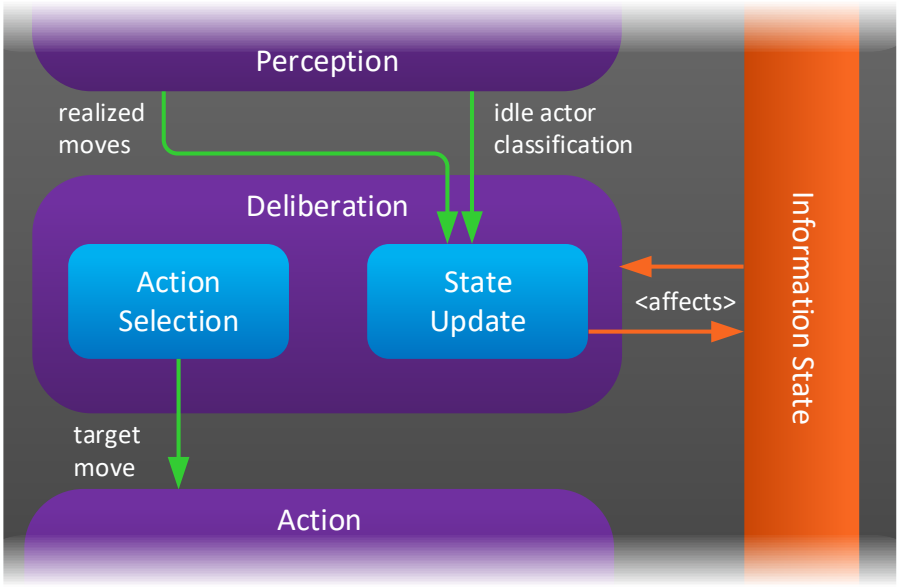


Figure 4.4: The deliberation stage of the agency process.

Were the realization of a move been an instantaneous event, we would need only deal with a single issue, namely: the recognition of an appropriate moment to execute it. However, seeing as that is not the case and that realization of moves is eventual, not only must we concern ourselves with finding appropriate moments for its initiation, but also recognition of times where it is more appropriate to outright cancel a move in progress.

To elaborate on that point: As the realization of a dialog move does not interfere with an actor's ability to perceive and process new information, both procedures may execute in tandem. Then, it might so happen that new information acquired yields a change of state which causes the move in progress to lose its status as an appropriate one. In some cases, the loss may be due to bad timing, and so an actor must halt and wait for the next opportune moment to re-initiate the move. In others, the actor's target move may have suddenly changed, in which case the actor must abandon its actual move altogether. Either way, cancellation is the result of a mismatch between what an actor thinks it ought to do and that which it is doing currently.

More formally: Let $M = \{m_0, m_1, m_2, \ldots, m_n\}$ be the set of all available dialog moves, with m_0 being the special *idle* move — indicating inaction — and $\{m_1, m_2, m_3, \ldots, m_n\}$ remaining arbitrary for now. In addition, let $m_t \in M$ refer to the *target* move an actor wishes to realize at this time, and $m_a \in M$ refer to the *actual* move currently being realized. Lastly, define the predicate $AT(m)$ to be

Symbol	Description
m_0	The dialog move representing idle action.
m_t	The target dialog move intended to be realized.
m_a	The actual dialog move being realized.
$AT(m)$	A predicate over dialog moves which evaluates to truth iff m is an appropriate move to make at this time.

Table 4.1: Symbols used in describing action timing.

Algorithm 4.1 Timing the realization of dialog moves.

1: **if** $m_a \neq m_t$ **then**
2: $\quad m_a \leftarrow m_0$ ▷ **Cancel** the move in progress (bad content).
3: **end if**
4: **if** $m_t \neq m_0$ **and** $AT(m_t)$ **then**
5: $\quad m_a \leftarrow m_t$ ▷ **Realize** the intended move.
6: **else if** $m_a \neq m_0$ **and** $\neg AT(m_a)$ **then**
7: $\quad m_a \leftarrow m_0$ ▷ **Cancel** the move in progress (bad timing).
8: **end if**

true if and only if $m \in M$ is an appropriate move to make at this time; a proposition whose truth value is evaluated using a combination of the parameter m as well as the actor's state. Given that, we are now able to frame an actor's timing of moves in the intuitive routine listed in Algorithm 4.1.

The routine's principal role is to both detect and reconcile the aforementioned disparity between intention and actuality, whenever it occurs. In other words, it ensures that an actor either executes the target move when appropriate (i.e. $m_a = m_t$ when $AT(m_t)$ evaluates to true) or nothing at all (i.e. $m_a = m_0$ when $AT(m_t)$ evaluates to false or when $m_t = m_0$). In either case, the routine concludes with m_a having been assigned a sensible value.

There is however one caveat: as it stands in Algorithm 4.1, we operate under the hidden assumption that the idle move m_0 is appropriate at all times. Conversely, the same idea can be expressed as the acknowledgment that at a time when no move is known to be appropriate, the idle move is an acceptable 'out' from the dilemma. We argue this assumption is justified due to the unmitigated scarcity of any practical examples that contradict it.

Finally, once it is determined what move is to be realized, it is expressed to the outside world through behavior (Figure 4.5).

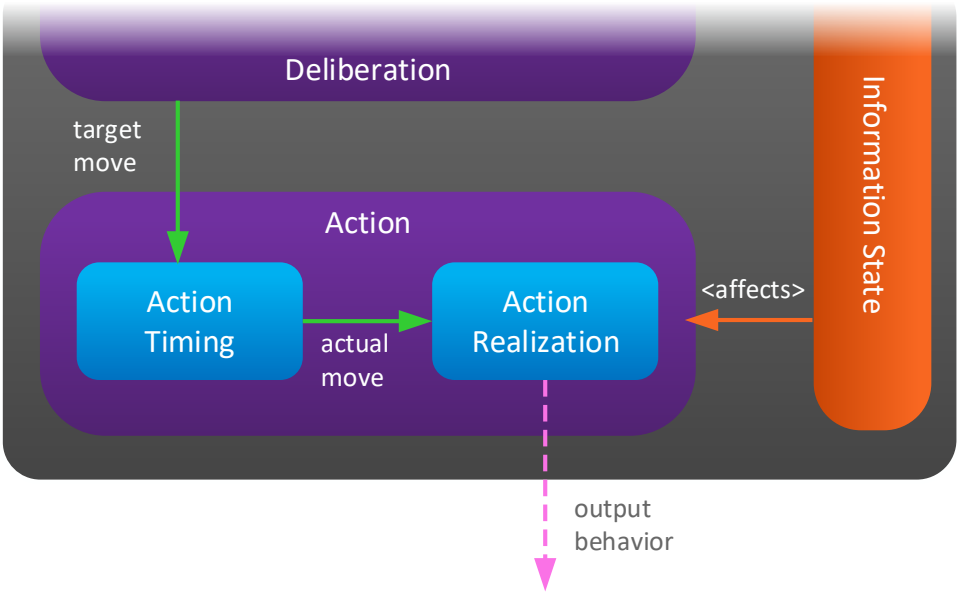


Figure 4.5: The action stage of the agency process.

4.2 Problem definition

As we have shown, a complete system of agency should not be viewed as a single entity, but rather as a collection of smaller sub-systems; with each being in charge of a single operation, and each being only loosely coupled to the rest. In the previous section, we have already presented a partial segmentation of the agency process into steps that would qualify as (largely) mutually independent sub-systems with single-operation responsibility — but only up to a point.

For example, we did make some universally applicable observations regarding the dialog move selection step, but we have not committed ourselves to any concrete mechanism which would satisfy it. The reason we avoid concertizing steps of the agency process further than we already have is that each represents a whole field of inquiry on its own, to which the best answers are either unknown as of yet, or scenario-dependent, or both.

Acknowledging that, our position is that the best available avenue for progress consists of parallel independent research and experimentation: An approach where it is possible for different contributors to take up separate steps of the agency process to work on independently, and whose proposed mechanisms for those steps are eventually merged together to form a complete agency system.

To this end, a standardized baseline framework embodying the universal traits of the agency process

would come a long way. Its chief goals would then be the following two:

1. Provision of a reasonable segmentation of the agency process into concrete, well-defined steps.
2. Provision of a mechanism by which these steps may be merged together into a complete working system.

We translate these goals into more detailed requirements in the next section.

4.3 Requirements

In the previous section we noted two goals a framework for agency should uphold: (1) a segmentation of the agency process into concrete steps, and (2) a mechanism by which these steps may be glued together. In this section, we break these goals down to more specific requirement descriptions:

1. The framework **must** represent the agency process in its entirety, such that when all implemented steps have been merged together, they form a complete working system.

2. The framework **must** model individual steps of the agency process as separate from each other, in the sense that an implementation of one need not necessarily be aware of its counterpart in another. This allows for users to combine contributions from separate sources into one system without mandating developers of those contributions to be aware of each other's work.

3. The framework **must** provide a baseline representation for dialog moves. That is, one which is generic enough so that it does not limit a users' freedom in deriving from it whatever concrete moves their particular use-case demands.

4. The framework **must** allow for the data stored within the information state to be of arbitrary type, so that users will have maximal freedom in choosing its preferred representation on a case-by-case basis.

5. The framework **must** provide a mechanism through which it would be possible to access and mutate data in the information state from within steps of the agency process. This, in order to allow for output (or intermediate results) of one step to be accessible from another without failing the second requirement.

In the next section we outline a framework design that fulfills these requirements.

4.4 Architecture

In this section we specify the agency system framework's design in detail, where we cover the following topics: The segmentation of the agency process into modules (sections 4.4.1–7), information state structure (Section 4.4.8), and representation of dialog moves (Section 4.4.9).

4.4.1 Operation modules

At the base of the framework is a segmentation of the agency process into a set of *operation modules* that closely mirror the same components of the process that were detailed in sections 4.1.3–5 (also visible in Figure 4.6). Each module bears responsibility for the execution of a single operation, is (largely) independent from the rest, and has well-defined input/output parameters. Together, an organized execution of these modules is what would form a complete agency system.

There are a total of six modules, below they are categorized under the three main stages of agency introduced in Section 4.1:

- **Perception**
  - **Recent activity perception:** Perceives realized dialog moves.
  - **Current activity perception:** Classifies actors as idle/non-idle.
- **Deliberation**
  - **State update:** Updates the information state based on perceived actor activity.
  - **Action selection:** Designates a target move to make.
- **Action**
  - **Action timing:** Decides whether the target move is appropriate at this time.
  - **Action realization:** Expresses moves through behavior.

Each module is responsible for the execution of exactly one operation in accordance with given input and the current information state, and is expected to produce appropriate output in return. In the following sections 4.4.2–7, we provide an extended description for all six modules, wherein we elaborate on their respective:

- Operation description.
- Input/output specification.

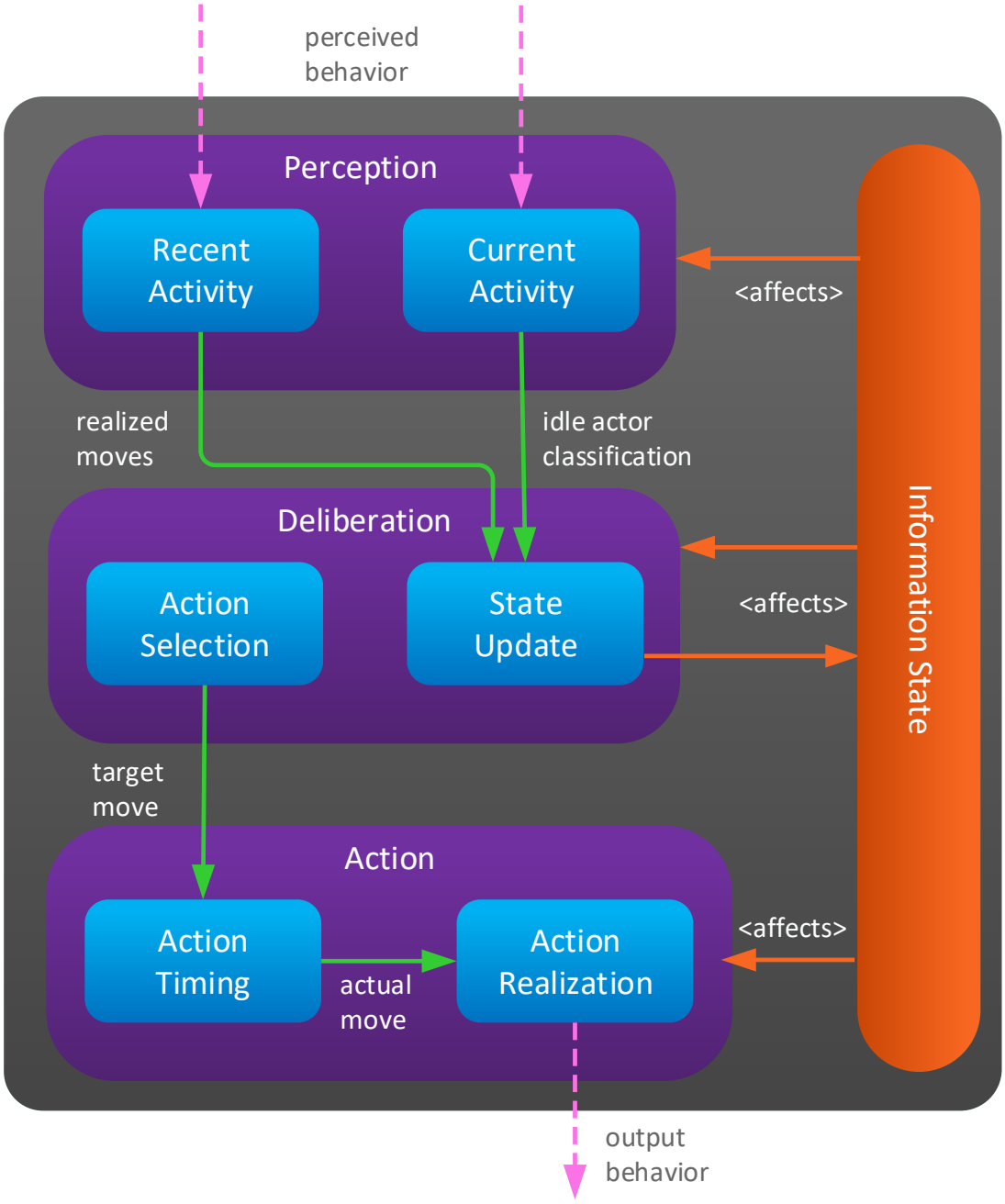


Figure 4.6: An overview of the complete agency process.

- Invocation order in relation to other modules.
- State-related permissions.

On a related note: Some modules have their input/output denoted simply as 'custom' and without further details. That is the case when that input/output involves concrete actor behavior. As it is discussed in Section 4.1, the exact form behavior takes depends on details which are beyond the scope of this framework, and therefore it remains amorphous from our perspective. An exact specification of behavior-related parameters is ultimately an issue that is best addressed by the relevant module's implementer, which is the reason we mark input/output involving it as custom-made.

4.4.2 Module: Recent activity perception

Description	Determines the set of moves that have taken place recently.
Input	*<custom>*
Output	Set of dialog move realization events.
Invocation index	0
State access	Read-only

The recent activity perception module interprets the perceived behavior of other actors in the scene as dialog moves of the near past. That is, in the time since this module's latest previous invocation and up to its current one. Together with the current activity perception module (Section 4.4.3), it is the first to be invoked during a process iteration. Its output is a set of move realization events (pairings of dialog moves with the actors who performed them) and is fed into the state update module (Section 4.4.4).

Note on output When the output set contains multiple moves, they are to be understood as to have taken place simultaneously. This is due to the implicit assumption that the time between successive iterations of the actor's agency process is short enough to justify the 'simultaneous' label.

Note on output The output set should not contain the idle move. Actors who are not present in the set will be assumed idle by the framework.

4.4.3 Module: Current activity perception

Description	Classifies actors as either passive (idle) or active (non-idle).
Input	*<custom>*
Output	Mapping of actors onto $\{passive, active\}$.
Invocation index	0
State access	Read-only

The current activity perception module interprets the perceived current behavior of other actors in the scene as either passive or active. That is, as actors who are either in the midst of realizing the idle move or not, respectively. Together with the recent activity perception module (section 4.4.2), it is the first to be invoked during a process iteration. Its output is fed into the state update module (Section 4.4.4).

Note on output The produced mapping serves a secondary purpose: its set of keys (actors) can be regarded as the total set of actors the system is aware of. Therefore, changes in this set over consecutive iterations of the agency process may be used to track actor (dis)engagement from the conversation.

4.4.4 Module: State update

Description	Determines and applies the effect perceived activity has on the information state.
Input	Combined output from both perception modules, plus the system's own activity snapshot.
Output	*<none>*
Invocation index	1
State access	Read/Write

The state update module evaluates perceived activity to determine and apply its effect on the information state. It is the first module to be invoked proceeding the conclusion of the perception stage, and is also the sole module with permission to mutate the state. Proceeding this module's execution and for the remainder of the agency process' current iteration, the newly updated state is that which would be presented to the action selection module (Section 4.4.5) and the modules of the action stage (sections 4.4.6 and 4.4.7). In the next iteration, it would be presented to both perception modules (sections 4.4.2 and 4.4.3) and the upcoming invocation of the state update module.

Note on input The system's own activity snapshot comes in the form of three dialog moves: the *recent move*, *target move*, and *actual move*. The recent move refers to the latest the system has completely realized, the target move to the one the system wishes to realize currently, and the actual move to the one that the system is actually realizing.

4.4.5 Module: Action selection

Description	Determines the target move to realize.
Input	*<none>*
Output	Dialog move.
Invocation index	2
State access	Read-only

The action selection module evaluates the system's state and decides on which dialog move to designate as a desirable one to make — we call it the 'target move', because it might differ from the one which would end up being realized in actuality. The module is invoked immediately following the conclusion of the state update module, and its output is fed into the action timing module (section 4.4.6).

Note on output If it is undesirable for the system to make any move, this module's output must be the idle move.

4.4.6 Module: Action timing

Description	Determines appropriate timing of the specified move.
Input	Dialog move.
Output	Boolean value.
Invocation index	3
State access	Read-only

The action timing module decides on the binary question: is now a good time to realize the specified move? It is invoked following execution of the deliberation stage, and its output indirectly implies the input to the action realization module (Section 4.4.7).

Note on output If the input move is the idle move, the output of this module is ignored — the framework operates under the assumption that the idle move is always an appropriate move to realize (this assumption and its justification is mentioned in Section 4.1.5).

4.4.7 Module: Action realization

Description	Realizes the specified move.
Input	Dialog move.
Output	*<custom>*
Invocation index	4
State access	Read-only

The action realization module expresses a specified dialog move through output behavior. It is the last module to be invoked in an iteration. Its input move is indirectly implied by the system's target move and the output of the action timing module (Section 4.4.6).

Note on input The input move is one of {target move, idle move}.

4.4.8 Information state

The information state object is accessible to all modules. The state is made up of individual data *components*. Components are simply containers of a single type of data, and a state may incorporate at most one component of each type.

During instantiation of a new system, modules can declare which components they require be available to them through the information state. If a system is instantiated with a state that does not satisfy the requirements of one or more modules, the framework terminates it with an error. This compels the system administrator to comply and ensures that when setting up the information state object to use (Section 4.5.2), it indeed supports all needed components.

During their invocation, modules may access each of the components available on the information state, and the state update module is even permitted to alter the values of data stored within them.

4.4.9 Dialog moves

The framework provides an abstract generic representation of dialog moves, one from which concrete moves can be derived. It is made up of a distinction between three types of moves: (1) zero-target, (2) single-target, and (3) multi-target. Zero-target moves are not addressed to any particular actor or group of actors (e.g. laughter, the idle move), single-target moves carry meaning intended for exactly one recipient (e.g. a hand-shake, name inquiry), and multi-target moves are aimed at one or more targets (e.g. greeting a crowd).

Users of the framework are able to take any of these basic archetypes, derive their own moves from it, and are permitted to extend it with additional properties as they see fit.

4.5 Usage

In this section we describe usage of the framework from the point of view of two potential user roles: the first is one that wishes to implement a module (Section 4.5.1), and the second is one that wishes to take existing module implementations and combine them into a working system (Section 4.5.2).

4.5.1 Module implementation

The module implementation work-flow is very straightforward, and can be expressed in four short steps:

1. Select which of the six modules to implement, and derive from its respective interface.
2. Specify which components the module requires be available on the information state.
3. Specify any initialization logic via a specialized method, to be overridden by the implementer.
4. Specify the logic driving the module's actual operation, and make sure that the module produces appropriate output.

4.5.2 System administration

From a system administrator's perspective, building a new working system instance is achieved like so:

1. Select six module implementations to use (one of each type).
2. Initialize a new information state object, and make sure to enable all components required by the selected modules and that each component assumes a sensible default value.
3. Create a new system instance with the chosen modules and state.
4. Advance the simulation by invoking iterations over the agency process.

4.6 Summary

In this chapter we outline a formulation of a framework for agency systems, intended to be used as a universal jump-start for agency modeling in conversational actors. It abstracts an actor's agency process into a collection of several distinct modules, and enables implementations of these modules from unrelated contributors to be easily joined back together to make up a complete working system. It makes no assumptions regarding the modalities of communication involved, nor the number of actors taking part in the interaction. In Chapter 5 we demonstrate its usage as we model a new approach to action selection/timing for actors, and in Chapter 6 we combine the result together with the communication management system from Chapter 3 in a prototype application as a proof of concept.

Chapter 5

Agency through social practice

In Chapter 4 we outline a framework modeling the actor-internal aspect of multi-modal/multi-party conversation, calling it 'the agency process'. We then proceed to segment that process into several distinct modules, the two most important of which are arguably the action selection and timing modules. The first decides on the semantic content an actor should communicate, and the second is responsible for identifying the right moment in time for that communication to take place. Up until now, we have only specified these insofar as their general responsibility and the intended format of their input/output. In this chapter, we go one step further and introduce a concrete mechanism by which both modules could be implemented.

Our proposed approach is inspired by social practice theory, and so we open the chapter with a brief introduction to that field and describe why (and how) social practices might be useful as a guide for an actor's behavior (Section 5.1). Next, we break down the larger problem of specifying a mechanism that would achieve this into several smaller questions (Section 5.2) divided among three main categories: (1) Describing social practices, (2) deriving action from them, and (3) using them to appropriately time said action (sections 5.3–5, respectively). Finally, we cover a brief evaluation of this approach regarding its capabilities/limitations (Section 5.6), and conclude with a summary (Section 5.7).

5.1 Domain overview

Social practice theory attempts to articulate the symbiotic relationship between the actions of social beings and the systematic rules (be they explicit or implicit) that govern their societies [23, 35, 40]. It garnered considerable interest in the seventies when well-respected sociologist Pierre Bourdieu studied and defined core concepts and mechanisms by which social practices may be argued about [5, 24, 29].

To Bourdieu, the idea that people make decisions based on a purely rational, finely laid-out reasoning seemed far-fetched and overrated. He rather believed that our actions spring out of a much more elementary rule-system based in practical — sometimes even unconsciously primal — logic.

In conversation, social practices dictate a great deal of our conduct: When greeted by an extension of the arm, we are expected to shake hands; During an encounter with authority we rather not speak unless spoken to first; while during a friendly group chat interruptions and overlapping speech are much more likely to happen. These are examples of how we let social practices dictate when and how we can communicate. It is true that their influence does not enter our conscious mind often, but it is undeniable that culture, customs, norms, and values play just as an important role in conversation as they do in all other aspects of our social lives. Therefore, we are lead to believe that viewing dialog from the perspective of social practices is a promising new angle for a solution to the problem of action selection/timing in multi-party conversation [9].

The idea of incorporating social practices in the development of agency systems brings with it some very appealing benefits. The first of which pertains to ease-of-use: For us, reasoning about social practices is naturally intuitive, because they are concepts of which we are very much aware and make use of in everyday life. This fact alone already opens the door towards development of tools and methods that would allow anyone to take part in agency system modeling — regardless of their degree of technical proficiency in programming or even computer science in general.

A second benefit to working with social practices is their one-size-fits-all quality with respect to the actors they encompass. By this, we mean to say that in theory, even though the set of actors participating in a social practice may be heterogeneous (i.e. containing actors with varying roles and/or governed by different expectations), an encoding of that single practice would contain nearly all information needed to control any of its members.

As the third and last selling-point of social practices, we note that their hierarchical nature easily yields itself to component reuse. Here, when we use the word ‘hierarchy’ we refer to the idea that practices of higher complexity can often be expressed as a composition of practices of lower complexity. With that in mind, it is entirely possible that a single lower-complexity practice plays a part in more than one of higher-complexity. In such cases, the shared practice need only be specified once, and from then on it is available for (relatively effortless) reuse.

5.2 Problem definition

In order to build a viable mechanism for controlling actor behavior using social practices, we break down the larger problem into several smaller ones spanning three main categories: (1) Formal description of practices, (2) derivation of rules for action selection from practice descriptions, and (3) the same as the latter, but for timing of selected action. These translate into the following research questions:

1. **What is a suitable model for describing practices?**
   (a) How could it support multi-party interactions?
   (b) How could it be used to track the progression of a practice in an ongoing interaction?
2. **How could such descriptions aid in action selection?**
   (a) How could they point to when it is more appropriate to not perform any action?
3. **How could such descriptions aid in action timing?**
   (a) How could they decide when to initiate interruption?
   (b) How could they decide in what way to respond to an interruption?

We explore our answer to these in order during the three sections 5.3–5 respectively, starting with our proposed method to formally describe social practices in the next.

5.3 Describing social practices

In this section we introduce an expectation-based model as our approach to the formal description of (conversational) social practices. First, we define exactly what we mean by the words ‘practice’ and ‘expectation’, make a distinction between several kinds of expectations, and demonstrate how they may be combined to describe whole practices (Section 5.3.1). Then, we explain how the dynamic progression of a practice can be tracked throughout an ongoing interaction (Section 5.3.2), and discuss in detail how that idea applies to each kind of expectation differently (sections 5.3.3–9).

5.3.1 Practices and expectations

In the context of conversation, we use the term *practice* to denote a socially-mandated pattern of interaction. For example: actors mutually greeting each other at the onset of dialog is one of the most basic instances of such a pattern. And even though practices may vary in complexity, context, and frequency

of occurrence, their common denominator is their assignment of various *expectations* at different times to either individual actors or groups.

Each and every expectation falls under one of two kinds: either *atomic* or *composite*. Atomic expectations predict the realization of concrete dialog moves by specific actors. We refer to the actor performing a move as its *source*, and to the pairing of a dialog move with its source as an *event*. For this reason, we use the terms event/atomic expectation interchangeably in the remainder of this chapter. Event expectations are specified further in Section 5.3.3.

In contrast with atomics, composites serve as containers for other expectations. We use the term *parent* to refer to the composite itself and *children* to refer to the collection of expectations it contains. Composites are used to indicate some temporal relationship amongst their children, and/or to impose logical constraint(s) upon them. In this chapter, we define six types of composite expectations: *sequential*, *conjunctive*, *disjunctive*, *divergent*, *repeating*, and *conditional*. These are specified further in sections 5.3.4–9, and an overview of the context-free grammar governing their arrangement (and including atomics) is available for viewing in Figure 5.1.

Using expectations and events, an instance of a practice can be formally represented by a three-tuple $\rho = (A, M, X)$, where A is the set of participating actors, M the set of possible dialog moves, and X an expectation-arrangement *program* recognized by the language defined in Figure 5.1. X itself is a single (composite) expectation that serves as the root ancestor of all others, and whose descendant atomics all exclusively refer to events in $A \times M$.

5.3.2 Expectation relevance and resolution

Before we move on to discuss specific types of expectation in detail, we want to first be able to relate a practice's expectation arrangement X to the dynamic progression of an ongoing interaction. To do this, we augment expectations of X with two kinds of state-annotation: *relevance* and *resolution*.

Relevance refers to the applicability of a given expectation to a given time. Remember: composite expectations such as X expresses a temporal relation between their children; This means that throughout the course of an interaction different expectations may be applicable at different times. We label those expectations that are currently applicable as *relevant/active* and those that are not as *irrelevant/inactive*. Note that the children of a composite expectation do not necessarily inherit its relevance status, but we defer that discussion until later sections, where we detail the behavior of individual expectation types themselves.

Accompanying the annotation for relevance is another for resolution. Recall that expectations are

<expectation list> → <expectation>

→ <expectation>, <expectation list>

<expectation> → <atomic expectation>

→ <composite expectation>

<atomic expectation> → event: (move: <dialog move>, source: <actor>)

<composite expectation> → <sequence>

→ <conjunction>

→ <disjunction>

→ <divergence>

→ <repeat>

→ <conditional>

<sequence> → sequence: {<expectation list>}

<conjunction> → conjunction: {<expectation list>}

<disjunction> → disjunction: {<expectation list>}

<divergence> → divergence: {<expectation list>}

<repeat> → repeat: {<expectation>}

<conditional> → if <condition> then: {<expectation>}

Figure 5.1: Context-free grammar for the arrangement of expectations.

merely a human-friendly way for representing patterns of interaction. In this context, we view interaction as a sequence of events $I = (e_1, e_2, \ldots)$ that take place while the expectation is relevant; The expectation itself can be regarded as an implicit definition of a set $E = \{I_1, I_2, \ldots\}$ whose members include all possible interactions that adhere to the constraints imposed by that particular expectation. At the onset of interaction $I = \varnothing$, and proceeds to grow as new events take place. The expectation's resolution status then answers the following question: Will it ever be the case that $I \in E$? If it is already the case that $I \in E$, then the answer is positive and we resolve the expectation with *satisfaction*. On the other hand, if no member of E is prefixed by I then it is certain that $I \notin E$ and we resolve the expectation with *failure*. We label expectations that are either satisfied or failed as *resolved*, and those that are neither as *unresolved/pending*.

In the upcoming sections 5.3.3–9 we go over individual expectation types in detail, and for composites we also specify how the relevance and resolution status of the parent expectation relates to that of its children.

5.3.3 Event expectations

Event expectations (also called atomics) represent a pattern of interaction containing a concrete event e, and serve as the basic building block for more complex composite expectations. Uniquely, this type of expectation cannot resolve as a failure, but rather remains pending until eventually becoming satisfied once e takes place.

5.3.4 Sequential expectations

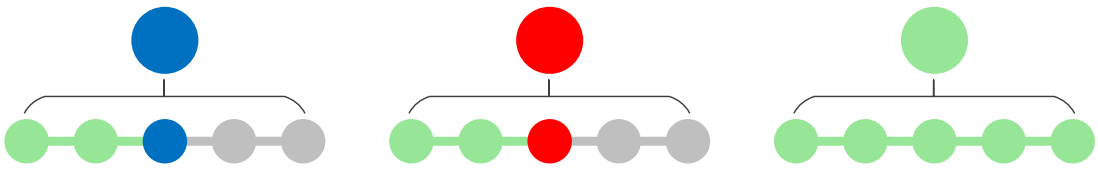
A sequential expectation p represents a pattern of interaction wherein a sequence of $n \geq 2$ child expectations $(x_1, x_2, \ldots, x_n)$ is satisfied one child at a time in the specified order. Consequently, at most a single child is active at a time. Let us identify the active child expectation by its index i in the sequence. At the onset of interaction $i = 1$, and it remains that way for as long as x_i is pending resolution. Once x_i is resolved, one of two scenarios unfolds: If x_i failed, then p fails as well. Otherwise, i is incremented by one unless $i = n$, in which case we have satisfied the entire sequence and therefore also p itself. This procedure is defined in Algorithm 5.1 and illustrated by Figure 5.2.

5.3.5 Conjunctive expectations

A conjunctive expectation p represents a pattern of interaction wherein $n \geq 2$ child expectations forming a set $\{x_1, x_2, \ldots, x_n\}$ are satisfied in any order. That is to say: all children are activated at

Algorithm 5.1 Evaluating the resolution status σ of a sequential expectation.

$\sigma \leftarrow$ *pending*
activate x_1; $i \leftarrow 1$ ▷ i identifies the active child.
while σ is pending **do**
    if x_i is pending **then continue**
    else if x_i has failed **then** $\sigma \leftarrow$ *failure*
    else if $i = n$ **then** $\sigma \leftarrow$ *satisfaction*
    else
        deactivate x_i; **activate** x_{i+1}
        $i \leftarrow i + 1$
    end if
end while ▷ σ is now assigned the resolution status of the entire expectation.



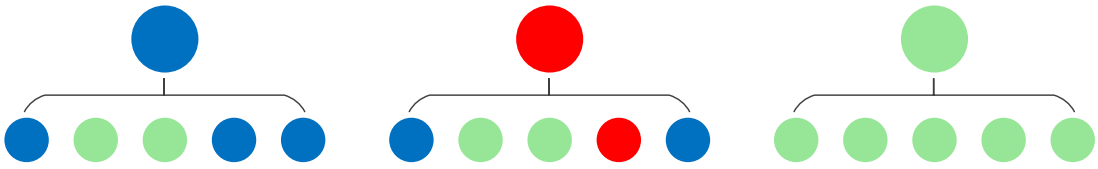
(a) (b) (c)

Figure 5.2: **Sequential expectations** and the relationship between resolution status of a parent and that of its children. (a) Pending resolution. (b) Failure of a child implies failure for the parent. (c) Satisfaction of all children in sequence implies satisfaction of the parent. Legend: pending expectations are in blue, failed in red, satisfied in green, and inactive in gray.

the onset of interaction, and remain so until resolved. If any child fails, then p fails, and only once all children have been satisfied will p be satisfied. This procedure is defined in Algorithm 5.2 and illustrated by Figure 5.3.

Algorithm 5.2 Evaluating the resolution status σ of a conjunctive expectation.

1: $\sigma \leftarrow$ *pending*
2: **activate** $x_1, x_2, \ldots x_n$
3: **while** σ is pending **do**
4: **if** $\exists i \in [1, n] : x_i$ has failed **then** $\sigma \leftarrow$ *failure*
5: **else if** $\forall i \in [1, n] : x_i$ is satisfied **then** $\sigma \leftarrow$ *satisfaction*
6: **end if**
7: **end while** ▷ σ is now assigned the resolution status of the entire expectation.



(a) (b) (c)

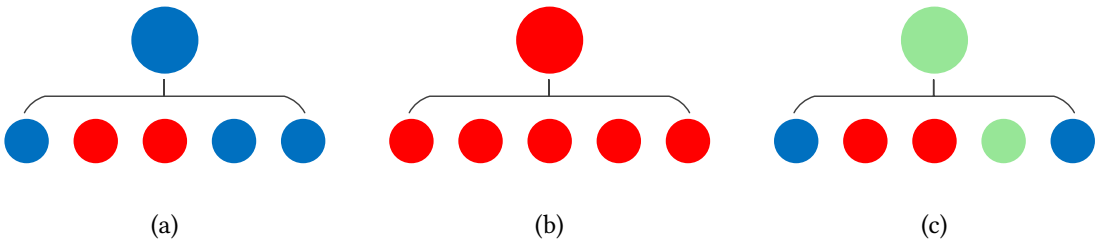
Figure 5.3: **Conjunctive expectations** and the relationship between resolution status of a parent and that of its children. (a) Pending resolution. (b) Failure of any child implies failure for the parent, regardless of the resolution status of other children. (c) Only the satisfaction of all children implies satisfaction of the parent. Legend: pending expectations are in blue, failed in red, and satisfied in green.

5.3.6 Disjunctive expectations

A disjunctive expectation p represents a pattern of interaction wherein at least one of a set of $n \geq 2$ child expectations $\{x_1, x_2, \ldots, x_n\}$ is satisfied. That is to say: all children are activated at the onset of interaction, and as soon as any one of them is satisfied so will be p. The only case where p fails is when all children fail. This procedure is defined in Algorithm 5.3 and illustrated by Figure 5.4.

Algorithm 5.3 Evaluating the resolution status σ of a disjunctive expectation.

1: $\sigma \leftarrow$ *pending*
2: **activate** $x_1, x_2, \ldots x_n$
3: **while** σ is pending **do**
4: **if** $\exists i \in [1, n] : x_i$ is satisfied **then** $\sigma \leftarrow$ *satisfaction*
5: **else if** $\forall i \in [1, n] : x_i$ has failed **then** $\sigma \leftarrow$ *failure*
6: **end if**
7: **end while** ▷ σ is now assigned the resolution status of the entire expectation.



(a) (b) (c)

Figure 5.4: **Disjunctive expectations** and the relationship between resolution status of a parent and that of its children. (a) Pending resolution. (b) Only the failure of all children implies failure of the parent. (c) Satisfaction of any child implies satisfaction for the parent, regardless of the resolution status of other children. Legend: pending expectations are in blue, failed in red, and satisfied in green.

5.3.7 Repeating expectations

A repeating expectation p represents a pattern of interaction wherein a single child expectation x is satisfied zero or more times. Consequently, this type of expectation cannot be satisfied, but rather remains perpetually pending unless there comes a time where x fails. This procedure is defined in Algorithm 5.4.

Repeating expectations can be combined with sequential ones to set the minimum number m of repetitions expected. This can be achieved by defining a sequential expectation whose first m children are x and whose last child is p. Likewise, to set an upper limit n on the number of repetitions, one could define a disjunction with two children where one of them is p and the other a sequence containing n times x. Lastly, in order to set both lower and upper limits, one may use a sequence whose first m children are x, and whose last child is a disjunction between p and another sequence containing $n - m$ times x.

Algorithm 5.4 Evaluating the resolution status σ of a repeating expectation.

1: $\sigma \leftarrow$ *pending*
2: **activate** x
3: **while** σ is pending **do**
4: **if** x has failed **then** $\sigma \leftarrow$ *failure*
5: **else if** x is satisfied **then reactivate** x ▷ Restores x to its original status.
6: **end if**
7: **end while** ▷ σ is now assigned the resolution status of the entire expectation.

5.3.8 Conditional expectations

A conditional expectation p represents a pattern of interaction wherein a single child expectation x is satisfied, but is only activated while a specified predicate $C(\ldots)$ evaluates to *true*. In other words, p remains pending while $C(\ldots) = false$, and is assigned the same resolution status as x otherwise. This procedure is defined in Algorithm 5.5.

5.3.9 Divergent expectations

A divergent expectation p represents a pattern of interaction wherein exactly one out of a set of $n \geq 2$ child expectations $\{x_1, x_2, \ldots, x_n\}$ is selected to be satisfied. However, before we could explain the selection procedure we must first introduce a new attribute of composite expectations: *scope carriers*.

Algorithm 5.5 Evaluating the resolution status σ of a conditional expectation.

$\sigma \leftarrow$ *pending*
while σ is pending **do**
    if $C(\ldots)$ is *true* **then activate** x
    else deactivate x
    end if
    if x is active **then**
        $\sigma \leftarrow$ resolution status of x
    end if
end while ▷ σ is now assigned the resolution status of the entire expectation.

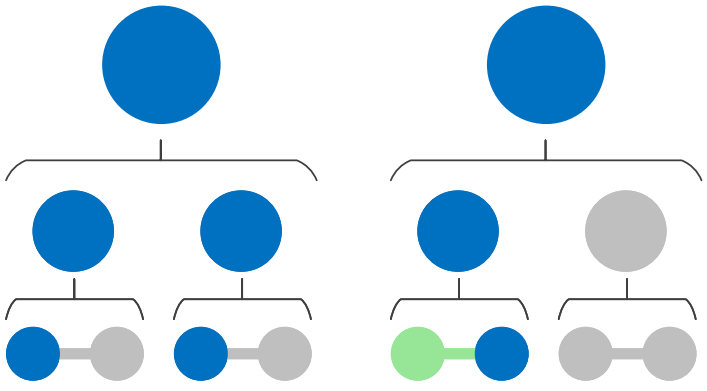
When a composite expectation has exactly one child currently active, this child is said to be its parent's scope carrier. It is so termed because the resolution status of both child and parent is dependent on the same scope: the child's. Among composites defined in this chapter so far, the following posses scope-carrier children: Sequences, repetitions, and possibly conditionals (when their predicate is *true*). When a given expectation x has scope carrier $s(x)$, that child may also in turn have its own scope carrier $s(s(x))$ and so on. We call the sequence $(x, s(x), s(s(x)), \ldots)$ the *scope carrier chain* of x, with x itself at the head and x's most distant scope-carrying descendant — call it $\hat{s}(x)$ — at the end. We can now define the aforementioned divergent child selection procedure using $\hat{s}$.

At the onset of interaction, p behaves in an identical manner to that of a disjunction, and continues to do so until $\hat{s}(x_i)$ is satisfied for some $i \in [1, n]$. At that point, all children are deactivated except for x_i (which — by the way — causes $s(p) = x_i$). From then on, p's resolution status is assigned the same value as that of x_i. This procedure is defined in Algorithm 5.6 and illustrated by Figure 5.5.

Ultimately, the divergent child selection procedure comes down to detecting the first child which has undergone some sort of partial satisfaction. Initially, we entertained the idea of defining partial satisfaction of an expectation x to mean the satisfaction of either x or any of its descendants. However, we were afraid that — due to the potential of some expectations' descendant set to be quite large — it might become difficult to reason about and/or hold a complete mental picture of a divergence. For this reason, we came up with the concept of scope carriers to restrain the number of expectations to be considered. We figure that further experimentation and practical experience is needed before a more definitive verdict could be issued on this matter.

Algorithm 5.6 Evaluating the resolution status σ of a divergent expectation.

$\sigma \leftarrow$ *pending*
$x_s \leftarrow$ *null* ▷ Hosts the (eventually) selected child.
activate $x_1, x_2, \ldots, x_n$
while σ is pending **do**
    if $x_s =$ *null* **then**
        if $\exists i \in [1, n] : x_i$ is satisfied **then** $\sigma \leftarrow$ *satisfaction*; **break**
        else if $\forall i \in [1, n] : x_i$ has failed **then** $\sigma \leftarrow$ *failure*; **break**
        else if $\exists i \in [1, n] : \hat{s}(x_i)$ is satisfied **then**
            $x_s \leftarrow x_i$
            deactivate all in $\{x_j \mid j \in [1, n],\ j \neq i\}$
        end if
    else
        $\sigma \leftarrow$ resolution status of x_s
    end if
end while ▷ σ is now assigned the resolution status of the entire expectation.



(a) (b)

Figure 5.5: **Divergent expectations** and scope carrier selection. Depicted is a divergence between two sequential expectations. (a) Preceding scope carrier selection. (b) Proceeding scope carrier selection. Legend: pending expectations are in blue, satisfied in green, and inactive in gray.

5.4 Selecting action

In the previous Section 5.3 we define social practices using a three-tuple $\rho = (A, M, X)$, where A is the set of participating actors, M the set of possible dialog moves, and X a program expressed in a domain-specific language made up of social expectations. In this section we show how to derive action from such descriptions of practices.

When deriving action from a practice ρ, it is always with respect to one actor $a \in A$. As an ongoing interaction progresses, expectations in X undergo changes of state including both their relevance and resolution (Section 5.3.2). By going through X's expectation tree and looking for active nodes, we can collect a set of events $E = \{e_1, e_2, \ldots, e_n\}$ that ρ implies are currently expected to be performed by members of A. We call E the *expected event set*. To extract from E those events which are relevant for the specific actor in our query, we apply a filter yielding: $E_a = \{e \mid e \in E,\ e\text{'s source is } a\}$. From here, we define the *candidate dialog move set* for a:

$$M_a = \begin{cases} \{m \mid m \text{ is referenced by some } e \in E_a\}, & \text{if } |E_a| \neq 0 \\ \{m_0\}, & \text{otherwise } (m_0 \text{ being the idle move}) \end{cases}$$

Be aware: this assumes that nothing is barring an actor from realizing any of the candidate moves at this point in time. This assumption generally works when the dialog moves in question do not require interaction with the environment, however in practice that may not always be the case (e.g. if a move realization requires a prop that is not currently in the actor's possession).

The next step is to pick one target move to perform out of the candidate set. When $|M_a| = 1$ this is trivial, but what about the case where $|M_a| \geq 2$? It seems we are back to square one, because once again we are faced with the same problem we set out to solve: How should one go about selecting a target move out of a given set? However, even though we did not achieve a general answer to the question, we did in fact manage to use social practices to significantly prune the search space; Recall that the set being referred to in the original problem was in fact M — the entire collection of available moves — whereas from our experimental experience so far (in Chapter 6), we gather that in practical applications M_a represents only a very small subset of that.

5.5 Timing action

In the previous Section 5.4 we show how a practice description $\rho = (A, M, X)$ can aid in the selection of a target move $m_t \in M$ to perform for a given actor in A. To complement that, we use this section to propose another mechanism through which the appropriate time to execute m_t can be decided upon.

Let us begin by reviewing the simplest cases first: When $m_t = m_0$ (with m_0 being the idle move) the problem is trivial, because as was previously discussed (in Section 4.1.2) the idle move is universally appropriate at all times. And so, now we are left to consider what should be done when $m_t \neq m_0$. Recall from the previous Section 5.4 that m_t was so chosen that it is guaranteed to be expected at the current time, in other words: content-wise m_t is definitely appropriate. This means that the only remaining source of potential objection to the target move's temporal validity would lie in the turn-taking aspect of conversation.

Define the *dialog floor* to be the set $F = \{a \mid a \in A, a \text{ is active}\}$ containing all actors who are currently in the process of realizing non-idle moves. Again, let us consider the easiest cases first: If $F = \varnothing$, then there is no reason why m_t should not be performed. The same is true for when $F = \{\text{Self}\}$, with Self being the actor whose action we are timing. Therefore it is when F contains one or more actors other than Self where things get challenging.

We view cases where other actors beside Self are active and Self has a pending move to perform as two sides of the same coin: On the one side, when Self $\notin F$ we decide on a potential *initiation of interruption*; on the other, when Self $\in F$ we decide on a potential *surrender to interruption*. We use the expectation arrangement in X to evaluate both decision scenarios.

Recall from our description of social practices (Section 5.3) that X is a tree of expectations, with atomics as its leaves and composites as its internal nodes. Also recall that as an interaction progresses, nodes of X go in and out of relevance. With each node of $x \in X$ we associate two sets of rules $R_i(x)$ and $R_s(x)$: the first concerning interruption-initiation and the second for -surrender. Each rule is a four-tuple made up of the following: (1) a precondition indicating whether the rule is currently applicable, (2) an indicator function signaling which actors of A the rule affects, (3) an implication value in $\{true, false\}$, and (4) a weight $w > 0$.

To answer either interruption scenario, we visit every active node in X and consult relevant rules from each. A rule is considered relevant when both its precondition holds and Self is affected by it. Each rule consulted casts a weighted vote in favor of its implication (either *true* or *false*). Once all rules have cast their virtual votes, the implication with the greater tally decides the answer. For potential interruption initiation, *true* implies the go-ahead to interrupt and *false* instructs to remain idle. For potential surrender, *true* implies Self should abort whatever move is being realized and *false* instructs to ignore other active actors and press on. For an overview of the action timing procedure in its entirety, see Algorithm 5.7.

Algorithm 5.7 Decide whether it is an appropriate time to perform the target move.

function TimeAction(m_t)

Input: The target move to perform.

Output: Either *do perform* or *do not perform*.

if $m_t = m_0$ **or** $F = \varnothing$ **or** $F = \{\text{Self}\}$ **then return** *do perform*

end if

$$R \leftarrow \begin{cases} R_i, & \text{if Self} \notin F \\ R_s, & \text{otherwise} \end{cases}$$

$X_a \leftarrow \{x \mid x \in X, x \text{ is active}\}$

$V \leftarrow \{r \mid \exists x \in X_a : r \in R(x), r \text{ is applicable and affects Self}\}$ ▷ Define the voter base.

$V_0 \leftarrow \{r \mid r \in V, r \text{ implies } \textit{false}\}$

$V_1 \leftarrow \{r \mid r \in V, r \text{ implies } \textit{true}\}$

$c_0 \leftarrow \sum_{r \in V_0} \text{weight of } r$

$c_1 \leftarrow \sum_{r \in V_1} \text{weight of } r$

if Self $\notin F$ **then**

▷ Decide whether to initiate an interruption.

return $\begin{cases} \textit{do perform}, & \text{if } c_1 \geq c_0 \\ \textit{do not perform}, & \text{otherwise} \end{cases}$

else

▷ Decide whether to surrender to an interruption.

return $\begin{cases} \textit{do not perform}, & \text{if } c_1 \geq c_0 \\ \textit{do perform}, & \text{otherwise} \end{cases}$

end if

end function

5.6 Discussion

The mechanisms outlined in this chapter are a first step towards social-practice-based agency. However, there are still many avenues left open for improvement and this section elaborates on what we regard as the principal two: (1) Supplementing practice descriptions with additional expectation types, and (2) prospective improvements to the proposed methods for action selection and timing. We cover these topics in their respective order in sections 5.6.1 and 5.6.2.

5.6.1 Inventing additional expectation types

The current collection of expectations specified in this chapter represents those that we thought were elementary to the description of any conversational scenario. Indeed, many common patterns of interaction can be expressed as a combination of these elementary types. That being said, we are certain there is plenty of room for additions.

One category that could clearly use a boost is atomics. Currently, it fields a single expectation implying a pattern of interaction wherein an event is to take place at some point in the future (Section 5.3.3). This type of expectation has its uses, but can be too general to accurately model interactions involving strict protocols of communication. For an increased level of control, expectations need to be defined that imply an event (or a set of events) is to take place either immediately or during another specifically-determined stretch of time.

The second significant problem we must address is the lack of attention to expectation failure. This is directly tied to the lack of additional atomics mentioned above. Since the only atomic currently specified is by definition immune to failure, it is virtually impossible to build potentially-failing expectation arrangements. Additionally, even if it were possible we are still lacking mechanism(s) capable of handling them. For example: a composite that activates a specified child when another fails would come a long way in mending this issue. Originally we did begin work on expectation types that would address these concerns, however due to time constraints we were unable to complete their specification prior to the publishing of this document.

5.6.2 Action selection and timing

In addition to the challenges mentioned previously, two other areas receptive to improvement are our proposed methods for action selection and timing. As was noted in Section 5.4, we have successfully managed to utilize social practices to significantly prune the search space for possible action. And yet, we remain stuck without a concrete procedure for deciding on the final choice. We are in agreement

with previous work [9] when we say that social practices alone can only bring us so far, and that the most reasonable approach to the final decider should be goal-oriented. That is to say: we believe that a promising approach would be the development of a reasoning-engine capable of considering each potential action, looking ahead to predict its effect on individual expectations, and selecting the one that most closely aligns with the actor's personal desires.

As for action timing, we feel that our current approach is overly complex. Specifically, we believe that some aspects of the rules governing interruption can be simplified (for example, the removal of a rule's precondition predicate), but we are unable to commit to any alteration without extensive real-world experience with the current specification.

5.7 Summary

In this chapter, we outline a new approach to agency inspired by social practice theory. We use the concept of expectations as a building block for practice descriptions, and then specify mechanisms through which both action and its timing can be derived from such descriptions. In addition, we also discuss the method's current limitations and prospective avenues for improvement. In Chapter 6, we provide an example of social-practice-based agency in a virtual (multi-party) couples-therapy scenario, where it is used not only to control the actions of artificial actors but also to present appropriate dialog options to a human player.

Chapter 6

Results

In chapters 3 and 4 we propose two tools to be used as facilitators in the modeling of multi-modal/multi-party conversation, respectively: (1) a system covering the actor-external aspect of the simulation, which is the management of inter-actor communication; and (2) a framework modeling the actor-internal aspect, which is the basic agency process taking place within individual actors themselves. Then, in Chapter 5 we used the agency framework to develop a new social-practice-based approach to action selection and timing. In this chapter, we demonstrate how merging all of this work together results in a viable multi-modal, multi-party application scenario.

We open with a description of the simulated scenario and an overview of the application architecture (Section 6.1). Next, we discuss the role each of the three main components — the communication management system, the agency model, and social practices — plays in our particular simulation scenario (sections 6.2–4, respectively), and finally we close the chapter with a summary (Section 6.5).

6.1 Overview

In this section we provide a short description of our chosen simulation scenario (Section 6.1.1) and familiarize ourselves with the application's general structure (Section 6.1.2).

6.1.1 Scenario description

The scenario we chose to simulate is that of a short couples-therapy session with three participating actors: one therapist and a patient couple. We note the following three attributes of this particular setting as the principal motivators for our choice: (1) It contains the minimal number of actors required for an interaction to be considered multi-party; (2) the set of participating actors is clearly heteroge-

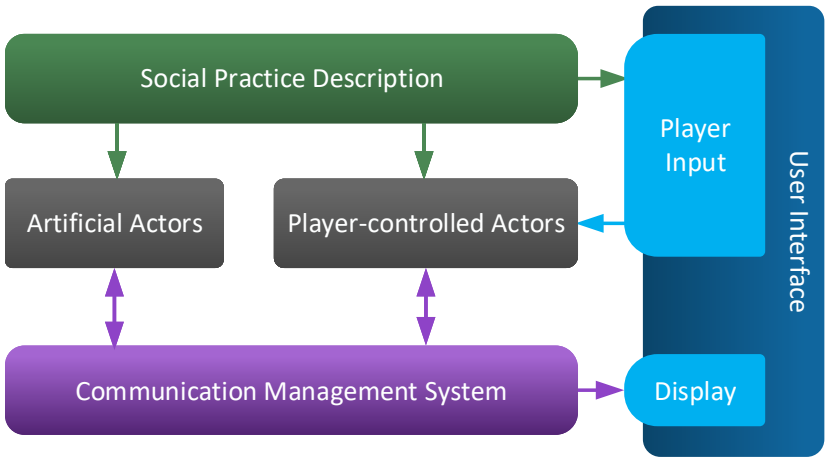


Figure 6.1: An overview of the application's architecture.

neous, that is to say not all actors fulfill the same role; and (3) it involves an interaction whose flow is — relative to most day-to-day conversations — quite strict and therefore more easily expressed using social practices.

In our particular implementation, the patient couple behavior is driven artificially, while the therapist is controlled by a human player. This choice was made bearing in mind that one of our main goals for multi-party simulation is the training of medical professionals in dealing with tough, emotionally-charged conversations.

6.1.2 Application architecture

To realize the chosen simulation scenario, we make use of the tools and methods developed in chapters 3 to 5. Figure 6.1 illustrates how they all relate and interact with each other, and also where the human player fits in.

At the heart of the application is an instance of the communication management system (CMS) from Chapter 3. Whenever an actor outputs some data representing its behavior, that data is submitted to the CMS, which synchronizes its broadcast to all other engaged actors. In our case, these are the two artificially-driven patient couple, the player-controlled therapist, and a silent *eavesdropper* created by the application's user-interface (UI) component to listen in on the conversation so that it may be visualized on the screen.

All actors besides the UI's eavesdropper have their behavior influenced by a single social practice description. It details the expected content and timing of actions for each actor in a couples-therapy

session, and is expressed using the concepts and methods from Chapter 5.

Actors themselves are internally structured according to the agency process from Chapter 4. Recall that it consists of one information state and six modules dividing the process into three stages: (1) perception, (2) deliberation, and (3) action. In our simulation, perception-stage modules are fed behavior-encoding data from the CMS, and extract from it information about the state of other actors and their actions. Next, modules of the deliberation stage use the processed data to update the actor's internal copy of the practice description and to re-evaluate the resolution status of the expectations it contains. Once done, an appropriate action is selected, and when the time is right also converted to behavior by action-stage modules. Finally, the produced output is submitted to the CMS for dissemination.

We use the next three sections 6.2–4 to expand on the role of each of the three major components mentioned, respectively: the communication management system, the agency process of different actors, and the scenario's social practice.

6.2 Role of the communication management system

Recall from Chapter 3 that the communication management system (CMS) is a central coordinator of read/write access to a batch of data channels, where each channel carries the data encoding one communication modality. Ideally, actors read the data off the CMS and use their own internal methods of interpretation to deduce the dialog moves — i.e. semantics — being broadcast. However in the interest of keeping the demo minimally simple, we bend the rules a little bit.

In our simulation, the CMS carries two data channels: one for speech (represented by strings of text), and another for dialog moves. By having a channel carrying them directly, we nullify the need for dialog moves to be interpreted from the text of emitted utterances (in line with the work of v. Oijen & Dignum [2012]). We wish to emphasize that this was done solely to simplify the demo's implementation, and that a production-ready system would most likely not be able to take the same shortcut. That being said, there are applications where the interpretation step can be rightly omitted, for example when a user selects the dialog move to perform from a menu rather than through interpretation of text or other modalities, as is often the case in serious games.

6.3 Role of the agency process

Recall from Chapter 4 a division of the agency process into three stages: (1) perception, (2) deliberation, and (3) action, with the output of each stage being fed as input to the next. In sections 6.3.1–3 we outline

Figure 6.2: An actor remains classified as active for a brief period of time proceeding its latest speech. This to account for natural pauses between consecutive utterances.

our implementation of each in order.

6.3.1 Perception stage

The implementation of perception for a simple simulation as ours is quite straightforward. The two tasks at hand: (1) recognition of recently-performed dialog moves and (2) classification of actors as either passive (idle) or active (non-idle). The former requires no work whatsoever, since the set of recent moves equals the current contents of the dialog move data channel of the communication management system. The latter is only slightly more challenging: we classify actors as active if and only if they have output some speech in the last t seconds, where t is predetermined to approximate the maximal pause duration between consecutive utterances in natural language. We call t the *activity grace period*, during which an actor retains its active status even though it may be idle. Figure 6.2 illustrates this concept.

6.3.2 Deliberation stage

Using the results of the perception stage, we proceed to update components of the actor's information state, these are: the passive/active classification record and the internal copy of the scenario's social practice description. The former consists of a simple setting of a variable, and the latter means we process the set of recently-performed moves to determine their effect on the relevance/resolution status of social expectations.

Proceeding the update, we use the new state of the expectation arrangement to select a dialog move to perform. Recall that the practice's expectations yield a set M_a of candidate moves. For artificial actors — in our use-case, the patients — we choose one member of M_a at random. For the therapist, we present M_a to the player and let him/her select a candidate. The screen-capture in Figure 6.3 showcases the

menu presented to the player.

6.3.3 Action stage

With a target move selected, only two tasks remain: action timing and realization. Our implementation of action timing is that which we defined in Chapter 5, and thus based on predetermined turn-taking rules we associated with individual sections of the scenario's social practice description. As for realization: To emphasize the non-discrete nature of speech and to allow for interruption, an actor realizing a move m with associated speech text $T(m)$ outputs $T(m)$ to the speech channel of the CMS one word at a time. We delay output of the next word by (approximately) the duration of time it takes to read the previous one. This helps simulate actual human speech. Once the entirety of $T(m)$ has been output, only then do we post m itself to the dialog move channel. This is done to simulate the fact that a dialog move can only be recognized once it has been realized fully. Figure 6.4 illustrates this output pattern.

6.4 Role of social practices

Recall from Chapter 5 that a social practice description ρ is defined as a three-tuple (A, M, X), where A is the set of participating actors, M the set of possible dialog moves, and X an expectation arrangement made up of events, sequences, conjunctions/disjunctions, and so on... To demonstrate how these components manifest themselves in a live application, we use this section as a walkthrough of the practice description for our couples-therapy scenario.

First things first: let $A = \{\text{PatientA, PatientB, Therapist}\}$ (in the application itself, the patient couple are named Alice and Bob while Charlie is the player-controlled therapist). Next, let M be the collection of dialog moves listed in Table 6.1 (with the implicit addition of the universal idle move):

Next, we elaborate on the structure of X — the practice's expectation arrangement. In the interest of brevity, let us denote an event (a realization of a move, see Section 5.3.1) with source s, move m, and target set $T = \{t_1, t_2, \ldots\}$ as $s : m \to T$. Now, we can begin our description of the couples-therapy scenario in terms of social expectations.

6.4.1 Expectation arrangement

Begin with the arrangement's root node — call it Session — which is a sequential expectation segmenting the entire interaction into three parts: Greetings, Counseling, and Goodbyes (Algorithm 6.1). Greetings and Goodbyes are both conjunctive expectations, each detailing an exchange of respectively

Figure 6.3: At any point in time the player may bring up the dialog move menu, which showcases the current candidate moves available to the therapist.

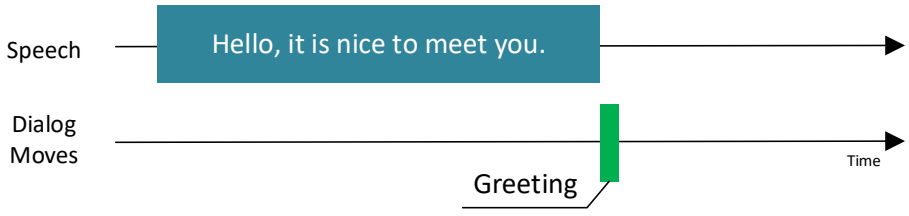


Figure 6.4: The realization pattern of a given dialog move begins with the output of associated speech text, followed by the move itself.

Move Type	Details	Performer
Acknowledgement	Affirmation of attention to the current speaker(s).	Any
Greeting	—	Any
Goodbye	—	Any
SessionClosing	Declaring the end of the counseling session.	Therapist
IssueSharingInvitation	Invitation to share an issue with one's partner.	Therapist
AdviceDispensation	Dispensation of advice in response to a shared issue.	Therapist
IssueSharing	Elaboration on an issue with one's partner.	Patient
IssueSharingDeclination	Declination to an issue-sharing invitation.	Patient

Table 6.1: A listing of the available dialog moves for the simulated scenario.

a Greeting/Goodbye between each patient and the therapist (Algorithm 6.2). We use a conjunction here to signal that it is not important in what order the moves are exchanged, as long they all do.

Algorithm 6.1 The sequential expectation at the root of a couples-therapy session scenario.

```
1: function Session
2:     expect sequence
3:         Greetings
4:         Counseling
5:         Goodbyes
6:     end sequence
7: end function
```

The Counseling expectation is a bit more complex. Within it, we wish to represent the following interaction: First, the therapist invites one of the patients to share an issue he/she is having with their partner. Then, the patient either accepts the invitation and elaborates on an issue, or declines. This process repeats until either all patients decline any further discussion of issues or the therapist decides it is time to end the counseling session. Algorithm 6.3 shows how a disjunction between the SessionClosing event and the IssueDiscussion cycle signals that the therapist may end the session at any time, and how a divergence is used to ensure exactly one issue discussion is ongoing at a time.

In addition to the therapist closing the session, we still need to model the case where neither patient wishes to discuss issues any further. To do this, we wrap the contents of any IssueDiscussion with a conditional expectation referencing a special bit-flag: open(p), indicating the a patient p is open to

Algorithm 6.2 An exchange of greetings between patients and therapist.

```
1: function Greetings
2:     expect all
3:         for each patient p do
4:             expect  p : Greeting → {Therapist}
5:             expect  Therapist : Greeting → {p}
6:         end for
7:     end conjunction
8: end function
```

Algorithm 6.3 The main flow of the counseling phase.

```
1: function Counseling
2:     expect any
3:         expect repeat
4:             expect one of
5:                 IssueDiscussion(PatientA)
6:                 IssueDiscussion(PatientB)
7:             end divergence
8:         end repeat
9:         expect  Therapist : SessionClosing → {}
10:    end disjunction
11: end function
```

discussion of his/her issues. Initially, open$(p) = true$ for all patients, and is only set to $false$ when a patient p declines an invitation to discuss an issue. This interaction pattern is visible in Algorithm 6.4, while the details of acceptance/declination of an IssueSharingInvitation are listed in algorithms 6.5 and 6.6.

Algorithm 6.4 Discussing an issue with a patient p.

```
1:  function IssueDiscussion(p)
2:      expect if open(p)
3:          expect sequence
4:              expect  Therapist : IssueSharingInvitation → {p}
5:              expect one of
6:                  IssueSharingAcceptance(p)
7:                  IssueSharingDeclination(p)
8:                  ▷ If p declines the invitation, then: open(p) ← false
9:              end divergence
10:         end sequence
11:     end conditional
12: end function
```

Algorithm 6.5 A patient p shares an issue.

```
1:  function IssueSharingAcceptance(p)
2:      expect sequence
3:          ▷ Acknowledging the invitation from IssueDiscussion(p):
4:          expect  p : Acknowledgement → {}
5:          expect  p : IssueSharing → {}
6:          expect  Therapist : Acknowledgement → {}
7:          expect  Therapist : AdviceDispensation → {}
8:          expect  p : Acknowledgement → {}
9:      end sequence
10: end function
```

Algorithm 6.6 A patient p declines to share an issue.

```
1: function IssueSharingDeclination(p)
2:     expect sequence
3:         ▷ Declining the invitation from IssueDiscussion(p):
4:         expect  p : IssueSharingDeclination → {}
5:         expect  Therapist : Acknowledgement → {}
6:     end sequence
7: end function
```

6.4.2 Interruption rules

Accompanying the expectation arrangement from the previous section are sets of interruption rules, each associated with one or more expectations. To keep things simple, we limit such associations to the higher-level nodes of the practice description: Greetings, Goodbyes, and Counseling. These rules affect the artificially-driven actors in the scene exclusively, while the player retains his/her freedom to perform dialog moves without delay and at any point in time.

The rules governing Greetings and Goodbyes represent indifference to conflict (Table 6.2a), that is to say actors perform their target moves immediately and to completion. On the other hand, rules governing the Counseling phase represent conflict-avoiding behavior, where actors avoid interrupting others and surrender the floor when they are interrupted themselves (Table 6.2b). In effect, this rule assignment allows for overlapping speech during the socially-lax parts of the interaction (greetings and goodbyes), but implies a more submissive stance from patients towards the therapist during actual counseling.

Interruption Initiation	
Precondition	*none*
Affects	*any*
Implication	*true* (do interrupt)
Interruption Surrender	
Precondition	*none*
Affects	*any*
Implication	*false* (do not surrender)

(a) Conflict-indifference.

Interruption Initiation	
Precondition	*none*
Affects	*any*
Implication	*false* (do not interrupt)
Interruption Surrender	
Precondition	*none*
Affects	*any*
Implication	*true* (do surrender)

(b) Conflict-avoidance.

Table 6.2: Two sets of rules used to regulate interruption in different phases of the interaction: (a) is used during Greetings, Goodbyes and (b) during Counseling.

6.5 Summary

In this chapter we outline how the tools and methods developed in previous chapters may be combined to simulate a multi-modal, multi-party couples-therapy session scenario[1]. First, we show how the communication management system from Chapter 3 is employed in synchronization of multiple layers of communication data. Secondly, we demonstrate that the agency system framework from Chapter 4 adapts to a specific use-case without necessitating excessive effort on our (the scenario administrator's) part. And finally, we substantiate the claim that social practice descriptions (Chapter 5) are capable of both driving action selection/timing in artificial actors as well as offering acceptable dialog controls to a human player.

[1]Download link: `https://tinyurl.com/rharel-gmt-thesis-2017-demo`

Chapter 7

Conclusion

In this brief closing chapter we look back at our original objectives for this research (Section 7.1) to evaluate where we succeeded and where we failed, and entertain ideas for future extensions of our work (Section 7.2).

7.1 Reflecting on research objectives

Recall our main research objective from Chapter 1, reproduced below:

> *Development of a **robust**, **generic** procedure for the construction of **multi-modal**, **multi-party** virtual conversation simulations.*

The multi-modal part of the objective refers to the interface through which data representing the different modalities in a scene is collected from one actor and redistributed to the rest. We believe the communication management system from Chapter 3 is an adequate fulfillment of this aspect. It supports an arbitrary amount of participating actors and modalities. Plus, it is flexible enough to accommodate both turn-based as well as real-time applications.

In addition to the communication management system, the multi-party component of the objective is also covered by the agency system framework from Chapter 4. Therein, we extract and package the domain-independent parts of virtual conversation in the form of several manageable modules, each with clearly defined input/output and responsibilities. To thoroughly and more accurately evaluate its usefulness additional experimentation is needed, but from our limited practical experience so far (in chapters 5 and 6), the core goals have been met.

To solidify robustness and generality, in Chapter 5 we develop a new method for action selection/-timing. By using aspects of social practice as our modeling tools, we presume this new approach is both

more scalable and intuitive to work with than prevalent information-state-based ones, such as Trindikit (see Section 2.1) [16]. Currently, we do not believe we have accumulated sufficient practical experience to determine whether this presumption is vindicated, but from what little we do have (Section 6.4) there is definite potential.

7.2 Directions for future work

The most pressing issue to be investigated is how well the developed tools and methods fair in practical applications. In Chapter 6 we build a small demo to showcase a proof-of-concept, but it is undoubtable that there is room for amendments to both the theoretical models used and their implementation. To better assess what specificities need adjustment, more comprehensive simulations for the envisioned target domain — virtual conversation in entertainment, education, and training of professionals — need to be developed and assessed.

Speaking of assessment, the evaluation of dialog systems is a notoriously difficult task. Ideally, the feedback received for a given iteration of a system in development should be as accurate as possible. Additionally, the stretch of time between a request for feedback and its reception should be as short as possible. However, these two requirements are hard to satisfy at once: If obtained through user studies, evaluation results are more accurate but take longer time and effort to compile. On the other hand, using automated tests and heuristics the delay in feedback becomes negligible but at the same time output is much harder to analyze than a human-written account of the system's performance. Any progress in this area is sorely needed.

The above concludes what we perceive to be the technically-oriented directions for future work. As for ones which are more exploratory in nature, we point to the social-practice-based approach to agency from Chapter 5. We believe it has substantial growth potential, particularly in the refinement of existing social expectation types and addition of new ones (see Section 5.6 for details). Once again, the only sensible way forward here is relentless experimentation and accumulation of practical experience.

References

[1] James F Allen et al. "Toward conversational human-computer interaction". In: *AI magazine* 22.4 (2001), p. 27.

[2] Jacob Aron. "How innovative is Apple's new voice assistant, Siri?" In: *New Scientist* 212.2836 (2011), p. 24.

[3] Dan Bohus and Eric Horvitz. "Decisions about turns in multiparty conversation: from perception to action". In: *Proceedings of the 13th international conference on multimodal interfaces*. ACM. 2011, pp. 153–160.

[4] Dan Bohus and Eric Horvitz. "Multiparty turn taking in situated dialog: Study, lessons, and directions". In: *Proceedings of the SIGDIAL 2011 Conference*. Association for Computational Linguistics. 2011, pp. 98–109.

[5] Pierre Bourdieu. *Outline of a Theory of Practice*. Vol. 16. Cambridge university press, 1977.

[6] *Databases*. The Association for the Advancement of Affective Computing. URL: `http://emotion-research.net/databases` (visited on 06/2016).

[7] Iwan De Kok and Dirk Heylen. "Multimodal end-of-turn prediction in multi-party meetings". In: *Proceedings of the 2009 international conference on Multimodal interfaces*. ACM. 2009, pp. 91–98.

[8] Jan Peter De Ruiter, Holger Mitterer, and Nick J Enfield. "Projecting the end of a speaker's turn: A cognitive cornerstone of conversation". In: *Language* (2006), pp. 515–535.

[9] Virginia Dignum and Frank Dignum. "Contextualized Planning Using Social Practices". In: *Coordination, Organizations, Institutions, and Norms in Agent Systems X*. Springer, 2014, pp. 36–52.

[10] Stan Franklin and Art Graesser. "Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents". In: *International Workshop on Agent Theories, Architectures, and Languages*. Springer. 1996, pp. 21–35.

[11] Agustı n Gravano and Julia Hirschberg. “Backchannel-inviting cues in task-oriented dialogue.” In: *INTERSPEECH*. 2009, pp. 1019–1022.

[12] Agustı n Gravano and Julia Hirschberg. “Turn-yielding cues in task-oriented dialogue”. In: *Proceedings of the SIGDIAL 2009 Conference: The 10th Annual Meeting of the Special Interest Group on Discourse and Dialogue*. Association for Computational Linguistics. 2009, pp. 253–261.

[13] Dušan Jan et al. “A computational model of culture-specific conversational behavior”. In: *Intelligent virtual agents*. Springer. 2007, pp. 45–56.

[14] Adam Janin et al. “The ICSI meeting corpus”. In: *Acoustics, Speech, and Signal Processing, 2003. Proceedings.(ICASSP'03). 2003 IEEE International Conference on*. Vol. 1. IEEE. 2003, pp. I–364.

[15] Tatsuya Kawahara, Takuma Iwatate, and Katsuya Takanashi. “Prediction of Turn-Taking by Combining Prosodic and Eye-Gaze Information in Poster Conversations.” In: *INTERSPEECH*. Citeseer. 2012, pp. 727–730.

[16] Staffan Larsson and David R Traum. “Information state and dialogue management in the TRINDI dialogue move engine toolkit”. In: *Natural language engineering* 6.3&4 (2000), pp. 323–340.

[17] Staffan Larsson et al. “TRINDIKIT 2.0 Manual-revised version”. In: *Trindi. Deliverable D* 5 (2000).

[18] Kornel Laskowski, Jens Edlund, and Mattias Heldner. “A single-port non-parametric model of turn-taking in multi-party conversation”. In: *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*. IEEE. 2011, pp. 5600–5603.

[19] Cheong-Jae Lee et al. “Recent approaches to dialog management for spoken dialog systems”. In: *Journal of Computing Science and Engineering* 4.1 (2010), pp. 1–22.

[20] Peter Ljunglöf. “trindikit. py: An open-source Python library for developing ISU-based dialogue systems”. In: *Proc. of IWSDS* 9 (2009).

[21] Lorenza Mondada. “Multimodal resources for turn-taking pointing and the emergence of possible next speakers”. In: *Discourse studies* 9.2 (2007), pp. 194–225.

[22] Joost van Oijen and Frank Dignum. “Agent communication for believable human-like interactions between virtual characters”. In: *International AAMAS Workshop on Cognitive Agents for Virtual Environments*. Springer. 2012, pp. 37–54.

[23] Sherry B Ortner. *Anthropology and social theory: Culture, power, and the acting subject*. Duke University Press, 2006.

[24] John Postill. “Introduction: Theorising media and practice”. In: *Theorising media and practice* (2010), pp. 1–10.

[25] Richard JD Power and Maria Felicita Dal Martello. “Some criticisms of Sacks, Schegloff, and Jefferson on turn taking”. In: *Semiotica* 58.1-2 (1986), pp. 29–40.

[26] David V Pynadath and Stacy C Marsella. “PsychSim: Modeling theory of mind with decision-theoretic agents”. In: *IJCAI*. Vol. 5. 2005, pp. 1181–1186.

[27] Anand S Rao, Michael P Georgeff, et al. “BDI Agents: From Theory to Practice.” In: *ICMAS*. Vol. 95. 1995, pp. 312–319.

[28] Antoine Raux and Maxine Eskenazi. “A finite-state turn-taking model for spoken dialog systems”. In: *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics. 2009, pp. 629–637.

[29] Renate Recke. “Outline of a Theory of Practice”. In: *Res Cogitans* 2 (2011), pp. 167–192.

[30] Harvey Sacks, Emanuel A Schegloff, and Gail Jefferson. “A simplest systematics for the organization of turn-taking for conversation”. In: *language* (1974), pp. 696–735.

[31] Johan Schalkwyk et al. ““Your Word is my Command”: Google Search by Voice: A Case Study”. In: *Advances in Speech Recognition*. Springer, 2010, pp. 61–90.

[32] Ethan O Selfridge and Peter A Heeman. “Importance-Driven Turn-Bidding for spoken dialogue systems”. In: *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics. 2010, pp. 177–185.

[33] Mei Si, Stacy C Marsella, and David V Pynadath. “Thespian: Modeling socially normative behavior in a decision-theoretic framework”. In: *Intelligent Virtual Agents*. Springer. 2006, pp. 369–382.

[34] Tanya Stivers et al. “Universals and cultural variation in turn-taking in conversation”. In: *Proceedings of the National Academy of Sciences* 106.26 (2009), pp. 10587–10592.

[35] *The Balance of Practice*. University of PA. URL: `http://www.elizd.com/website-LeftBrain/essays/practice.html`.

[36] Jason D Williams et al. “Rapidly scaling dialog systems with interactive learning”. In: *Natural Language Dialog Systems and Intelligent Assistants*. Springer, 2015, pp. 1–13.

[37] Margaret Wilson and Thomas P Wilson. “An oscillator model of the timing of turn-taking”. In: *Psychonomic bulletin & review* 12.6 (2005), pp. 957–968.

[38] Zerrin Yumak and Nadia Magnenat-Thalmann. “Multimodal and multi-party social interactions”. In: *Context Aware Human-Robot and Human-Agent Interaction*. Springer, 2016, pp. 275–298.

[39] Zerrin Yumak et al. “Modelling multi-party interactions among virtual characters, robots, and humans”. In: *Presence: Teleoperators and Virtual Environments* 23.2 (2014), pp. 172–190.

[40] Andrea Zhok. “Towards a theory of social practices”. In: *Journal of the Philosophy of History* 3.2 (2009), pp. 187–210.

List of Figures

List of Tables

List of Algorithms

Appendix A

Communication management system manual

This document is an instruction manual for the software library associated with the actor communication management system outlined in Chapter 3. It contains a listing of prerequisites (Section A.1), installation instructions (Section A.2), a brief walk-through of principal functionalities (Section A.3), and pointers to further documentation (Section A.4).

A.1 Prerequisites

- **C Sharp** is the programming language used across the project.
- **Visual Studio** is the integrated development environment used to manage source files and configure builds. Version 2015 and above should be compatible.

A.2 Installation

Currently, we do not host any pre-compiled DLLs for download, so you would want to build the library and its dependencies from source. The relevant directories in the repository are `core/dialog-communication/` and `core/commons/`. `dialog-communication` is the project directory of the system itself, while `commons` contains all sorts of utilities which we rely on as a dependency.

A.2.1 Building from source

In order to build all relevant projects from source, we recommend doing the following (all mentioned paths are relative to `core/dialog-communication/`):

1. Open the system's test project solution `test-dialog-communication.sln` located under `test/` in Visual Studio.
2. Build the solution.
3. Run all tests present through Visual Studio's test explorer to make sure they pass successfully (optional).
4. Both `dialog-communication.dll` as well as `commons.dll` should now be available under `test/bin/{Debug|Release}/`, depending on your desired build configuration. You may now reference both in your own projects as you wish.

A.3 Usage

General usage of the library follows a three step plan: (1) initialization, (2) actor engagement, and (3) invocation of the global update routine. We cover each of these in order throughout the remainder of this section.

A.3.1 Initialization

Recall from Section 3.4 that at the architecture's center is a single object: the communication manager. Also recall from Section 3.5.1 that the data types a manager is intended to support must be specified during its initialization. Suppose the actors in our scene communicate through two modalities called `Foo` and `Bar`. In this case, the manager's initialization code would resemble listing A.1.

```
class Foo { ... }
class Bar { ... }

var manager = new CommunicationManager.Builder()
    .Support<Foo>()
    .Support<Bar>()
    .Build();
```

Listing A.1: Initializing the manager.

A.3.2 Actor engagement and data subscription

Once we possess an initialized instance of a manager, we may begin engagement with actors (listing A.2). Firstly, note that in order for an object to engage with the manager, it must conform to the `ManagedActor` interface provided under the `Dialog.Communication.Actors` name space. Secondly, notice that an engagement results in a new data subscription being created. This fresh `subscription` object is the gateway through which the engaged actor's data preferences may be set.

```
class VirtualHuman : ManagedActor { ... }

var alice = new VirtualHuman(...);
DataSubscription subscription = manager.Engage(alice);
```

Listing A.2: Engaging an actor.

For example, if Alice were interested in communications of type `Foo`, she may indicate that in her subscription preferences alongside a callback which is to be invoked once such communications become available for viewing (listing A.3). And, if at some point in the future Alice is no longer interested in data of this type, she is free to alter her subscription (listing A.4) or disengage from the conversation entirely (listing A.5).

```
DataSubscription.Handler<Foo> foo_handler =
    (manager, packets) => { ... };

subscription.Include(foo_handler);
```

Listing A.3: Including a data type in a subscription.

```
subscription.Exclude<Foo>();
```

Listing A.4: Excluding a data type from a subscription.

```
manager.Disengage(alice);
```

Listing A.5: Disengaging an actor.

A.3.3 The update routine

Given a manager with one or more engaged actors, we may now invoke its global update routine (Algorithm 3.2 in Section 3.4.2) with a single method call, as per listing A.6. At this point, let us assume that an actor, Alice, is engaged with the manager and has the data type `Foo` included in her subscription preferences. Once invoked, the update routine will walk Alice through two steps: perception and action.

```
manager.Update();
```

Listing A.6: Invoking the update routine.

Recall from the previous section that beside the type specification itself, Alice also registered a callback method — called the *subscription handler* — to be associated with it, and which will now be used during the perception step. A subscription handler's signature can be seen in listing A.7, and contains three parameters: `T`, `manager`, and `packets` — with `T` being the targeted data type to handle (in our case, `Foo`), `manager` a reference to the communication manager invoking the handler, and `packets` an enumeration of packets carrying data of type `T`.

```
public delegate void Handler<T>
(
    CommunicationManager manager,
    IEnumerable<DataPacket<T>> packets
);
```

Listing A.7: Subscription handler method signature.

It is within the subscription handler(s) where the perception step takes place, and hence where actors get the chance to process and update their internal state based on the packets they are presented. Once all engaged actors have had their registered handlers invoked, the manager will commence the second update step: action.

```
public interface ManagedActor
{
    ...
    void Act(DataSubmission submission);
}
```

Listing A.8: Action method signature.

The action step is where actors are given the chance to produce their own data they wish be broadcast to others. It takes place within a special method — `Act()` — that all actors inherit from the `ManagedActor` interface (which they all share as per Section A.3.2). The signature of `Act()` is visible in listing A.8, and contains a single parameter: `submission`.

Just as a subscription object is used to regulate data transfer from the manager to actors, so is a submission object used to regulate data transfer from actors to the manager. One by one, the manager invokes `Act()` on all engaged actors, and supplies them with a submission object through which they may submit any data they wish. Following up on our example case: if Alice has some data she wishes be broadcast, she may do so by submitting it to the manager during an invocation of her own action method, as seen in listing A.9.

```
// <within Act()>

var some_foo = new Foo();
var some_bar = new Bar();

submission.Add(some_foo);
submission.Add(some_bar);
```

Listing A.9: Submitting data.

A.4 Further documentation

For more detailed documentation, we direct you to the complete reference site accessible through `core/dialog-communication/doc/html/index.html`. Alternatively, you may directly browse documentation comments in the source files themselves under `core/dialog-communication/library/`.

Appendix B

Agency system framework manual

This document is an instruction manual for the software library associated with the actor agency system framework outlined in Chapter 4. It contains a listing of prerequisites (Section B.1), installation instructions (Section B.2), usage guidelines for both operation module implementers as well as system administrators (sections B.3 and B.4, respectively), and pointers to further documentation (Section B.5).

B.1 Prerequisites

- **C Sharp** is the programming language used across the project.
- **Visual Studio** is the integrated development environment used to manage source files and configure builds. Version 2015 and above should be compatible.

B.2 Installation

Currently, we do not host any pre-compiled DLLs for download, so you would want to build the library and its dependencies from source. The relevant directories in the repository are `core/dialog-agency/` and `core/commons/`. `dialog-agency` is the project directory of the framework itself, while `commons` contains all sorts of utilities which we rely on as a dependency.

B.2.1 Building from source

In order to build all relevant projects from source, we recommend doing the following (all mentioned paths are relative to `core/dialog-agency/`):

1. Open the framework's test project solution `test-dialog-agency.sln` located under `test/` in Visual Studio.
2. Build the solution.
3. Run all tests present through Visual Studio's test explorer to make sure they pass successfully (optional).
4. Both `dialog-agency.dll` as well as `commons.dll` should now be available under `test/bin/{Debug|Release}/`, depending on your desired build configuration. You may now reference both in your own projects as you wish.

B.3 Operation modules

In this section we provide general instructions for the usage of the framework from a prospective module implementer's point of view. Starting with an overview of the module base types (Section B.3.1) and proceeding with details regarding module initialization (Section B.3.2), module-state interaction (Section B.3.3), and specific implementation instructions for each of the six module types (sections B.3.4–9).

B.3.1 Module bases

The first step in implementation of a new module, is to have it be derived from one of the six abstract module base types provided by the framework:

- `RecentActivityPerceptionModule`
- `CurrentActivityPerceptionModule`
- `StateUpdateModule`
- `ActionSelectionModule`
- `ActionTimingModule`
- `ActionRealizationModule`

Once we have a new module derived from one of the above, its functionality can be specified further by overriding several key methods provided by its base. Two of these are common across all module types, and pertain to a module's initialization logic and its interaction with the information state. These are covered in sections B.3.2 and B.3.3, respectively. The remaining methods pertain to each module's individual responsibility, and therefore are each tied to a specific base. These are covered in sections B.3.4–9.

B.3.2 Module initialization

Every module is initialized exactly once during its lifetime, immediately following construction of the system instance to which it belongs (Section B.4.3). At that time, it is given the opportunity to execute

any initialization-related logic it requires.

This is enabled by the `Setup()` method, which is available across all module base types through their shared ancestor, `class OperationModule<T>`. As is evident from listing B.1, usage of `Setup()` is completely optional, as its default implementation is a no-op.

```
public abstract class OperationModule<T>
{
    ...
    public virtual void Setup() { }
}
```

Listing B.1: Module setup method signature.

B.3.3 Module-state interaction

A module's interaction with the information state is relevant at two stages: (1) during initialization of a new system instance, when it is checked to see if the system's state object contains the necessary data required by the module, and (2) during the module's execution, when the module may read/write data from/to the state.

State data requirements

Different modules (and different implementations of modules) require different types of data be available to them through the state. While ensuring that the state indeed contains the appropriate data is the responsibility of a system's administrator (Section B.4.3), the framework does provide a mechanism by which a module can declare and enforce this type of requirements.

This is enabled by the `GetRequiredStateComponents()` method, which — again like in the case of `Setup()` — is available across all module base types. `GetRequiredStateComponents()` takes on a single parameter `types`: an initially empty collection, to be populated with all data types the module demands be available through the state.

For example, if we were to implement an action selection module whose logic requires data of type `Foo` to be available, we would do so in the manner seen in listing B.3.

If during construction of a new system instance there is an attempt made to pair a module with an information state that is not conforming to the module's requirements, the framework will terminate

```
public abstract class OperationModule<T>
{
    ...
    public virtual void
        GetRequiredStateDataTypes(ICollection<Type> types) { }
}
```

Listing B.2: Module state data requirement declaration signature.

```
class Foo { ... }

public sealed class MyActionSelectionModule : ActionSelectionModule
{
    ...
    public override void
        GetRequiredStateDataTypes(ICollection<Type> types)
    {
        types.Add(typeof(Foo));
    }
}
```

Listing B.3: Declaring state data requirements.

with an error. This mechanism helps enforce these requirements and enables system administrators to catch errors they made regarding state structure or state-module pairings at an early a stage as possible.

Mutable and immutable states

All modules have access to the system's information state through an inherited `State` property, and depending on the module's type, `State` takes on one of two forms: either mutable or immutable.

As is more thoroughly explained in Section B.4.3, the state is comprised of data components, with each component dedicated to holding a single type of data. Accessing these containers is allowed through both the mutable and immutable versions of the state. This is done through the `State.Get<T>()` method, with T being the type of data to retrieve. In contrast, only through the mutable version is it possible to overwrite components with new data. This is done through the `State.Set<T>(value)` method, with `value` being the new value to assign.

As is to be expected, the only module with access to the mutable state interface is the state update module. All other module types are confined to the read-only version.

B.3.4 Implementing recent activity perception

The recent activity perception module reports on recently occurring dialog moves and their sources. To implement this functionality, a module deriving from `RecentActivityPerceptionModule` must override the abstract method `ComposeActivityReport()` (listing B.4).

`ComposeActivityReport()` takes on a single parameter, `report`, through which we may submit whatever activity we perceive.

```
public abstract class RecentActivityPerceptionModule
    : OperationModule<ImmutableState>
{
    ...
    protected abstract void
        ComposeActivityReport(RecentActivityReport report);
}
```

Listing B.4: Recent activity perception module's main method signature.

Submission of an activity event is done by pairing a dialog move with the actor who performed it and adding the pair to the report via `report.Add(source, move)`, as can be seen done in

listing B.5. It is important to note that the report will not accept duplicates of events it already contains nor events whose dialog move is the idle move.

```
// <within ComposeActivityReport()>

Actor some_actor = ...;
DialogMove some_move = ...;

report.Add(source: some_actor, move: some_move);
```

Listing B.5: Reporting recent activity.

B.3.5 Implementing current activity perception

The current activity perception module reports the current activity status of actors, i.e. whether they are currently in the process of realizing a dialog move (active) or not (passive). Analogous to the implementation of a `RecentActivityPerceptionModule`, a module deriving from `CurrentActivityPerceptionModule` must override the abstract method `ComposeActivityReport()` (listing B.6), taking on a single parameter, `report`. We use the `report` object to submit our classification of actors.

```
public abstract class CurrentActivityPerceptionModule
: OperationModule<ImmutableState>
{
    ...
    protected abstract void
        ComposeActivityReport(CurrentActivityReport report);
}
```

Listing B.6: Current activity perception module's main method signature.

Classification of actors is done by marking them as either active or passive through the respective methods `report.AddAsActive(actor)` and `report.AddAsPassive(actor)` (listing B.7). It is important to note that the report will not accept duplicates of actors it already contains nor will it permit for an actor to be classified as both passive and active simultaneously.

```
// <within ComposeActivityReport()>

Actor some_passive_actor = ...;
Actor some_active_actor = ...;

report.AddAsPassive(some_passive_actor);
report.AddAsActive(some_active_actor);
```

Listing B.7: Reporting current activity.

B.3.6 Implementing state update

The state update module takes the combined input of the two perception modules together with a snapshot of the current state of the system, and bases a manipulation of the information state on those. A module deriving from `StateUpdateModule` must override the abstract method `PerformUpdate()` (listing B.8).

```
public abstract class StateUpdateModule
    : OperationModule<MutableState>
{
    ...
    public abstract void PerformUpdate
    (
        RecentActivityReport recent_activity,
        CurrentActivityReport current_activity,
        SystemActivitySnapshot system_activity
    );
}
```

Listing B.8: State update module's main method signature.

`PerformUpdate()` takes on three parameters: `recent_activity`, `current_activity`, and `system_activity`. The origin of `recent_activity` and `current_activity` is in the respective output of the two perception modules (sections B.3.4 and B.3.5, respectively), while `system_activity` is provided by the system itself.

The `recent_activity` report allows for inspection of its contents in the form of a collection of realized dialog move events stored in its `Events` property. Each *event* represents a pairing of a dialog move with the actor who made it (listing B.9).

```
// <within PerformUpdate()>

foreach (var @event in recent_activity.Events)
{
    Actor source = @event.Source;
    DialogMove move = @event.Move;
}
```

Listing B.9: Querying recent activity.

The `current_activity` report classifies actors as either active or passive, and makes this information accessible in two ways: (1) the `PassiveActors` property, containing a collection of all actors perceived as passive, (2) the `ActiveActors` property, which is the same for actors perceived as active, and (3) the `GetStatus(actor)` method, which yields an actor's current status flag: either `ActorActivityStatus.Passive` or `ActorActivityStatus.Active`. Example usage of the `current_activity` object is in listing B.10.

```
// <within PerformUpdate()>

foreach (var actor in current_activity.PassiveActors) { ... }
foreach (var actor in current_activity.ActiveActors) { ... }

Actor some_actor = ...;
var status = current_activity.GetStatus(some_actor);
if (status == ActorActivityStatus.Passive) { ... }
else if (status == ActorActivityStatus.Active) { ... }
```

Listing B.10: Querying current activity.

The `system_activity` snapshot is a small structure containing information about the system's own activity, expressed through three properties: `ActualMove` is the dialog move currently being realized, `TargetMove` is the move the system wishes to realize, and `RecentMove` is the most recently realized move (which in effect equals last update iteration's `ActualMove`).

```
public sealed class SystemActivitySnapshot
{
    ...
    public DialogMove RecentMove { get; }
    public DialogMove TargetMove { get; }
    public DialogMove ActualMove { get; }
}
```

Listing B.11: System activity snapshot structure.

Unlike all other module types, the `StateUpdateModule` is in fact the only one which has access to the state's mutator method `State.Set<T>(value)`. It is expected that an implementation of the module would take into account its input, process it in some manner, and update the values of relevant state data components accordingly. For more information regarding the information state object, see Section B.4.3, and for more about interaction with the state from within a module, see Section B.3.3.

B.3.7 Implementing action selection

The action selection module's job is to select a dialog move the system should aspire to realize. A module deriving from `ActionSelectionModule` implements this functionality by overriding the abstract method `SelectMove()` (listing B.12). `SelectMove()` takes no parameters, and is only responsible for returning the chosen dialog move.

```
public abstract class ActionSelectionModule
    : OperationModule<ImmutableState>
{
    ...
    public abstract DialogMove SelectMove();
}
```

Listing B.12: Action selection module's main method signature.

B.3.8 Implementing action timing

The action timing module helps coordinate the execution of a system's target dialog move. It does so by indicating whether a realization of that move at this time is acceptable. A module deriving from `ActionTimingModule` implements this functionality by overriding the abstract method `IsValidMoveNow()` (listing B.13). `IsValidMoveNow()` takes on a single parameter, `move`, representing the aforementioned target move, and is expected to indicate its appropriateness at this time with a corresponding boolean return value.

```
public abstract class ActionTimingModule
    : OperationModule<ImmutableState>
{
    ...
    public abstract bool IsValidMoveNow(DialogMove move);
}
```

Listing B.13: Action timing module's main method signature.

B.3.9 Implementing action realization

The action realization module is where dialog moves are converted into concrete behavior. A module deriving from `ActionRealizationModule` should override the abstract method `RealizeMove()` (listing B.14). `RealizeMove()` takes on a single parameter , `move`, representing a dialog move the system wishes be realized. The status of that realization is in turn indicated by yielding one of two flags: either `ActionRealizatioStatus.InProgress` or `ActionRealizationStatus.Complete`.

```
public abstract class ActionRealizationModule
    : OperationModule<ImmutableState>
{
    ...
    public abstract
        ActionRealizationStatus RealizeMove(DialogMove move);
}
```

Listing B.14: Action realization module's main method signature.

B.4 System administration

In this section we provide general instructions for the usage of a the framework from a prospective system administrator's point of view, wherein we cover four aspects of administration: (1) provision of actors and dialog moves (sections B.4.1 and B.4.2, respectively), (2) setting up of the information state (Section B.4.3), (3) instantiation of the system (Section B.4.4), and (4) simulation advancement (Section B.4.5).

B.4.1 Actors

Actors are represented by the `Actor` interface. It is a blank one, and serves primarily as a guarantor of type safety during the framework's internal operations. Regardless, objects which are to be regarded as actors must implement it, as it might take on additional properties and methods in a future iteration of the library.

B.4.2 Dialog moves

Dialog moves are represented by the abstract `DialogMove<TActor>` class, with `TActor` being the type of actor the move targets. For convenience, any concrete move derived from it may indicate its arity (i.e. the amount of targets it supports) through invocation of one of three possible base constructors, visible in listing B.15.

```
class MyActor : Actor { ... }

class MyMove : DialogMove<MyActor>
{
    public MyMove(DialogMoveKind kind)
        : base(kind) { ... }  // If untargeted.
    public MyMove(DialogMoveKind kind, MyActor target)
        : base(kind, target) { ... }  // If single-target.
    public MyMove(DialogMoveKind kind, IEnumerable<MyActor> targets)
        : base(kind, targets) { ... }  // If multi-target.
}
```

Listing B.15: Derivation of a new dialog move.

Administrators may formulate whatever moves their simulated scenario demands — with one ex-

ception: the special idle dialog move is already made available out of the box in the form of a singleton's property, `IdleMove.Instance`.

`DialogMoveKind` is an extendable enumeration provided so that it is more convenient to define the set of moves available in a scenario. Listing B.16 shows a sample dialog move kind definition.

```
class MyMoveKind : DialogMoveKind
{
    public static readonly DialogMoveKind
        Greeting = new MyMoveKind("greeting");
    public static readonly DialogMoveKind
        Goodbye = new MyMoveKind("goodbye");
}

// <usage>
Actor some_actor = ...;
var greeting = new DialogMove(MyMoveKind.Greeting, target: some_actor);
```

Listing B.16: Derivation of new dialog move kinds.

B.4.3 Setting up the information state

The information state is made up of what we dub as *data components*. Each component is associated with one type of data, and the state may support up to one component per type of data. As is explained in Section B.3.3, since the types of data a state is required to support are solely motivated by the operation modules comprising the system utilizing it, we opt to specify those during object construction.

For example, if our system contains modules that require data containers of type `Foo` and `Bar` be available, we are able to set up the state object to support these and at the same time initialize their respective components with a default value. This is illustrated in listing B.17.

B.4.4 Instantiating the system

A complete agency system is represented by the `AgencySystem` class. To instantiate it, two things are needed: (1) concrete instances of the six operation modules, and (2) a state object that conforms to their requirements. For more information regarding implementation of operation modules, see Section B.3; and regarding the information state object, see Section B.4.3.

```
class Foo { ... }
class Bar { ... }

var some_foo = new Foo(...);
var some_bar = new Bar(...);

var state = new InformationState.Builder()
    .Support<Foo>(initial_value: some_foo)
    .Support<Bar>(initial_value: some_bar)
    .Build();
```

Listing B.17: Information state construction.

Once we have both modules and a state, we can set up a new system instance in two quick steps: In the first, we wrap up the six operation modules in a `ModuleBundle` (listing B.18), and in the second we feed both the bundle and the state to the system's constructor (listing B.19).

```
var modules = new ModuleBundle.Builder()
    .WithRecentActivityPerceptionBy(my_RAP_module)
    .WithCurrentActivityPerceptionBy(my_CAP_module)
    .WithStateUpdateBy(my_SU_module)
    .WithActionSelectionBy(my_AS_module)
    .WithActionTimingBy(my_AT_module)
    .WithActionRealizationBy(my_AR_module)
    .Build();
```

Listing B.18: Bundling up modules.

B.4.5 Advancing the simulation

Advancing the simulation is a straightforward affair. Simply invoke `system.Step()` to perform a complete iteration of the system's agency process.

It is important to note that since both perception modules' input and the action realization module's output are left to be customized by the administrator, a recommended design approach for these would include some mechanism by which input/output can be transferred to/from them, and then to utilize that mechanism in the appropriate time with relation to calls of `Step()`. An illustration of this idea

```
ModuleBundle modules = ...;
InformationState state = ...;

var system = new AgencySystem(state, modules);
```

Listing B.19: Instantiating the system.

```
system.Step();
```

Listing B.20: Executing one iteration of the agency process.

is visible in listing B.21.

```
var RAP = new MyRecentActivityPerceptionModule(...);
var CAP = new MyCurrentActivityPerceptionModule(...);
var AR = new MyActionRealizationModule(...);
var system = new AgencySystem(...);

RAP.SetInput(...);  // Environmental input goes in.
CAP.SetInput(...);
system.Step();      // Input is processed.
AR.GetOutput(...);  // Actor behavior comes out.
```

Listing B.21: Coordinating custom module intput/output with system stepping.

B.5 Further documentation

For more detailed documentation, we direct you to the complete reference site accessible through `core/dialog-agency/doc/html/index.html`. Alternatively, you may directly browse documentation comments in the source files themselves under `core/dialog-agency/library/`.

Appendix C

Social-practice-based agency manual

This document is an instruction manual for the software library associated with the social practice based agency system outlined in Chapter 5. It contains a listing of prerequisites (Section C.1), installation instructions (Section C.2), usage guidelines (Section C.3), and pointers to further documentation (Section C.4).

C.1 Prerequisites

- **C Sharp** is the programming language used across the project.
- **Visual Studio** is the integrated development environment used to manage source files and configure builds. Version 2015 and above should be compatible.

C.2 Installation

Currently, we do not host any pre-compiled DLLs for download, so you would want to build the library and its dependencies from source. The relevant directories in the repository are `demo/dialog-spb/`, `core/dialog-agency/` and `core/commons/`. `dialog-spb` is the project directory of the system itself, `dialog-agency` is the directory of the framework (outlined in Chapter 4) on top of which the system is built, while `commons` contains all sorts of utilities which we rely on as a dependency.

C.2.1 Building from source

In order to build all relevant projects from source, we recommend doing the following (all mentioned paths are relative to `demo/dialog-spb/`):

1. Open the system's test project solution `test-dialog-spb.sln` located under `test/` in Visual Studio.
2. Build the solution.
3. Run all tests present through Visual Studio's test explorer to make sure they pass successfully.
4. All three DLLs: `dialog-spb.dll`, `dialog-agency.dll`, and `commons.dll` should now be available under `test/bin/{Debug|Release}/`, depending on your desired build configuration. You may now reference them in your own projects as you wish.

C.3 Usage

In this section we go over the available APIs for practice description (Section C.3.1), and cover the prepackaged action selection/timing module implementations (Section C.3.2).

C.3.1 Composing practice descriptions

Recall from Section 5.3 the specification of several expectation types used to describe practices. These are available in the form of distinct classes, one for each type. In addition, there is also a more user-friendly method-based interface that more closely resembles a natural language description of practices. In Table C.1 we list each expectation and its corresponding class- and convenience method names.

Detailed documentation for each class and method is available both in the source code as well as on the documentation site (see Section C.4 for details), so we do not cover that here. However, we do provide several usage examples of expectation arrangements using the natural-language-based convenience methods.

To start, we recommend importing all methods statically as demonstrated in listing C.1. Now, we can begin to arrange expectations together. Declaring an event expectation involves specification of the expected dialog move, and its source actor. This can be done through multiple variants, all visible in listing C.2. All of `Sequence`, `Conjunction`, `Disjunction`, and `Divergence` may have two or more children and therefore share the same usage pattern, listing C.3 illustrates it for conjunctive expectations, but the same form applies to the rest. `Repeat` is straightforward enough (listing C.4),

Expectation Type	Class Name	Convenience Method
Event	`EventExpectation`	`ExpectEvent`
Sequential	`Sequence`	`ExpectSequence`
Conjunctive	`Conjunction`	`ExpectAll`
Disjunctive	`Disjunction`	`ExpectAny`
Repeating	`Repeat`	`ExpectRepeat`
Conditional	`Conditional`	`ExpectIf`
Divergent	`Divergence`	`ExpectOne`

Table C.1: Expectation types and their corresponding class names and convenience methods.

and conditionals are similar to repeat except for an additional condition predicate parameter (listing C.5).

```
using static Dialog.SPB.Program.NLI.ProgramBuildUtilities;
// All Expect<...> methods are now in scope and ready to be used.
```

Listing C.1: Importing practice-building convenience methods into scope.

One last point to touch on is the ability to associate callbacks with different resolutions of an expectation. This feature can be accessed through any convenience method's optional `satisfied`, `failed`, and `resolved` parameters. Respectively, they can be used to associate pieces of logic with an expectation's satisfaction, failure, or either. An example usage is available for viewing in listing C.6.

```
// For zero-target moves:
ExpectEvent(kind: <move kind>, source: <actor>);
// For single-target moves:
ExpectEvent(kind: <move kind>, source: <actor>, target: <actor>);
// For multi-target moves:
ExpectEvent(kind: <move kind>, source: <actor>, targets: <actors>);
// For custom moves:
ExpectEvent(move: <move>, source: <actor>);
```

Listing C.2: Declaration variants of an event expectation.

```
ExpectAll
(
    "my conjunction",  // Expectations can be assigned titles.
    Expect..., Expect..., Expect...  // List child expectations.
);
```

Listing C.3: Declaration of a conjunctive expectation.

```
// Expecting the repeated satisfaction of a single child:
ExpectRepeat(Expect...);
```

Listing C.4: Declaration of a repeating expectation.

```
var some_flag = false;
ExpectIf(() => some_flag, Expect...);
```

Listing C.5: Declaration of a conditional expectation.

```
var some_flag = false;
Expect...(..., satisfied: _ => some_flag = true);
```

Listing C.6: Associating a callback with an expectation's resolution status.

C.3.2 Action selection and timing

Once we have described a social practice in terms of expectations, we pair that description with a specified actor to create a `SocialContext`. The actor in question is Self, i.e. the one representing the system itself. The next step is to store this social context as a component of the system's information state (Section 4.4.8). We do this so it will be accessible to two agency modules: one for action selection and another for timing (sections 4.4.5 and 4.4.6).

The action selection module is represented by the `SPBASModule` class (see Section 5.4 for details of operation), and requires no further setup other than specification of the winner candidate move selection method, whose signature is visible in listing C.7. Likewise, the action timing module is represented by the `SPBATModule` class. Its only input is a mapping between expectations of the social practice and the two sets of interruption rules mentioned in Section 5.5.

```
// Produces the index of the winning candidate.
public delegate int SelectionMethod
(
    List<DialogMove> candidates,
    ImmutableState state
);
```

Listing C.7: Winner candidate selection method signature.

C.4 Further documentation

For more detailed documentation, we direct you to the complete reference site accessible through `demo/dialog-spb/doc/html/index.html`. Alternatively, you may directly browse documentation comments in the source files themselves under `demo/dialog-spb/library/`.

Appendix D

Demo application manual

This document is an instruction manual for the application associated with the sample couples therapy scenario outlined in Chapter 6. It contains a listing of prerequisites (Section D.1), installation instructions (Section D.2), player control details (Section D.3), and pointers to further documentation (Section D.4).

D.1 Prerequisites

- **C Sharp** is the programming language used across the project.
- **Visual Studio** is the integrated development environment used to manage source files and configure builds. Version 2015 and above should be compatible.
- **Unity** is the game engine used to build the scenario, if you wish to inspect/alter application code you must install it. Version 5.6 and above should be compatible.
- **Visual Studio Tools for Unity** is not necessary but recommended if you wish to edit source code.

D.2 Installation

Currently, you may either browse online distributions at `https://tinyurl.com/rharel-gmt-thesis-2017-demo`, or build the executable from source. If you pursue the latter, then the relevant directories in the repository are `demo/application/example-couples-therapy/`,

and `demo/application/unity/`. `example-couples-therapy` is the project directory of the scenario model, and `unity` is that of the Unity project running it.

D.2.1 Building from source

In order to build the application from source, we recommend doing the following (all mentioned paths are relative to `demo/application/`):

1. Open the scenario's test project solution `test-example-couples-therapy.sln` located under `example-couples-therapy/test/` in Visual Studio.
2. Open Visual Studio's solution configuration manager and select `Release` for all projects except for `example-couples-therapy`, which should be assigned the `Release and Deploy to Unity` configuration.
3. Build the solution.
4. Verify that `unity/Assets/External/` contains the following DLLs (and if not, copy them over from the solution's `bin/Release/` directory):
   - `example-couples-therapy.dll`
   - `dialog-spb.dll`
   - `dialog-agency.dll`
   - `dialog-communication.dll`
   - `commons.dll`
5. Run all tests present through Visual Studio's test explorer to make sure they pass successfully.
6. Open `unity/` in the Unity editor and build the executable. Make sure `MainScenario` is the only scene included in the build.

D.3 Controls

Key	Function
P	Toggles the pause menu.
Space	Toggles the dialog move selection menu.

D.4 Further documentation

For more detailed documentation, we direct you to the complete reference site accessible through `demo/application/example-couples-therapy/doc/html/index.html`. Alternatively, you may directly browse documentation comments in the source files themselves under `demo/application/example-couples-therapy/library/`.