A Framework for Describing Communication Exercises in a Serious Game

An analysis, comparison and implementation.

Frank Wijmans

A thesis presented for the degree of
Master of Science

Universiteit Utrecht

Department of Information and Computing Sciences
University Utrecht
Utrecht
Netherlands
February 25, 2014

Abstract

This thesis investigates how we can simulate conversation and generate feedback for a serious game using a software framework, such that a player is given options and a non-playing character can react accordingly. An analysis of a communication training, given at the department of Pharmacy of Utrecht University, results in the functional requirements for software frameworks to simulate conversation. We describe conversations between two actors, interacting in various ways. We implement an example scenario such that we can compare frameworks based on dialog trees, belief, desire and intention models and domain reasoners on complexity, readability, usability, maintainability, and possibilities of generating feedback. The domain reasoner proves to be the best choice for defining exercises in communication training. We define an exercise in the IDEAS framework to see how we can best utilize the framework in the communication domain. While developing an exercise, we describe different ways of using rules and strategies.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Goals . . . 2

2 Analysing Pharmacy's Communication Training **4**
2.1 Theory . . . 5
2.1.1 Models . . . 6
2.1.2 Skills . . . 6
2.2 Practice . . . 8
2.2.1 Cases . . . 8
2.2.2 Conversation . . . 10
2.3 Discussion . . . 12
2.4 Goals of Communication Training . . . 12
2.5 Requirements for Defining Exercises . . . 13

3 Comparing Frameworks for Communication **15**
3.1 Preliminaries . . . 15
3.1.1 Haskell . . . 16
3.2 Technical Requirements for Software Frameworks . . . 16
3.2.1 Control . . . 16
3.2.2 Usability . . . 17
3.2.3 Maintainability and Adaptability . . . 17
3.2.4 Generating Feedback . . . 17
3.2.5 Concise Implementation . . . 18
3.3 Interactions of an Example Scenario . . . 18
3.3.1 Example: Miss Darcy Fetches Her Metformin 500mg . . . 19
3.4 Dialog trees . . . 20
3.4.1 Trees . . . 21
3.4.2 Root . . . 22
3.4.3 Edges . . . 23
3.4.4 Branches . . . 23
3.4.5 Overview . . . 24
3.5 Belief, Desires, and Intentions model . . . 26
3.5.1 Events . . . 26
3.5.2 Beliefs . . . 26
3.5.3 Desires . . . 27
3.5.4 Intentions . . . 28
3.5.5 Agents . . . 29

3.5.6 Conversation . . . 31
3.5.7 Overview . . . 31
3.6 Domain reasoners . . . 32
3.6.1 State . . . 33
3.6.2 Rules . . . 34
3.6.3 Strategy . . . 34
3.6.4 Overview . . . 36
3.7 Comparison . . . 36
3.7.1 Control . . . 37
3.7.2 Usability . . . 37
3.7.3 Maintainability and Adaptability . . . 38
3.7.4 Generating Feedback . . . 38
3.7.5 Concise Implementation . . . 38
3.8 Conclusion . . . 40

4 Designing Artificial Intelligence for 'Communicate!' 41
4.1 IDEAS framework . . . 41
4.1.1 Existing applications of IDEAS . . . 41
4.1.2 Paradigm . . . 42
4.2 Implementation . . . 43
4.2.1 Communication Exercises . . . 43
4.2.2 Conversation . . . 44
4.2.3 Sentences . . . 47
4.2.4 Strategies . . . 49
4.3 Design Choices . . . 51
4.3.1 Usage of the Strategy language . . . 51
4.3.2 Defining Sentences . . . 52
4.3.3 State Safety . . . 53

5 Conclusion and Future Work 55
5.1 Conclusion . . . 55
5.2 Future Work . . . 56
5.2.1 Context–dependent Rules . . . 56
5.2.2 Extended guards and Modifiers . . . 57
5.2.3 Introducing Knowledge System . . . 58
5.2.4 Analysis of Similar Communication Domains . . . 59
5.2.5 Exercise editor . . . 59
5.2.6 Validation . . . 60

List of Tables

2.1 Finding out how a client has used their medication. An example of closed questions in a conversation. 7

3.1 Comparison of three frameworks on control, usability, maintainability & adaptability and feedback generation. 37

3.2 Counts of data types and their constructors, and the number of helper functions and constructors used to define the example scenario. 39

List of Figures

2.1 Example of an case description card, used by a student to play the client's role. 9

3.1 Data type definition of interactions. 19
3.2 The first four options in the example scenario, defined using the *Interaction* constructor. 21
3.3 The structure of a dialog tree. 21
3.4 The core data types for describing conversation using a *DialogTree*. 22
3.5 The first option for a pharmacist is encoded in the root of the tree. 23
3.6 Encoding of the four options of the first choice for the pharmacist. 23
3.7 Encoding Branches for the players' second choice. 24
3.8 End–states for our example scenario in a dialog tree 24
3.9 A map of connected terms, from desires to events. 26
3.10 Definition of *Event* data type definition. 27
3.11 *Belief* defined as a logical language. 27
3.12 *Desire* data type definition and an example desire. 28
3.13 Function for building a weighted list, which encodes how strongly a desire is believed to be a goal. 28
3.14 The BDI definition of an agent. 29
3.15 Defining the example conversation for one of the agents in BDI framework. 30
3.16 Adding an option to the pharmacist's desires. 30
3.17 Definition of *Conversation* for a BDI framework between a player and a NPC 31
3.18 domain reasoners definition of state. 33
3.19 Some example state values. 34
3.20 Definition of a rule that wraps an interaction. 34
3.21 The first interactions of the example scenario. 35
3.22 Basic strategy data type definition for combining rules. 35
3.23 Defining a scenario strategy for the example. 35
3.24 Adapting a domain reasoner: adding part of conversation to the strategy. 36

4.1 An example domain reasoner in the IDEAS framework. 43
4.2 Example definition of a single exercise. 44
4.3 Definitions of the (data) types for the client parameters, learning goals and conversation knowledge. 44
4.4 Generalized *State* definition 45

4.5 Type definitions of the functions *toTerm* and *fromTerm* as part of the *IsTerm* class. 45
4.6 The *Conversation* data type as the term of the communications domain for pharmacy. 46
4.7 Helper–functions for guarding and modifying the Conversational Knowledge. 46
4.8 A helper–function that can modify the client's satisfaction parameter. 46
4.9 Type definition and constructor functions for *Sentence*s. 47
4.10 Definition of an example sentence; an interaction for a pharmacist. 48
4.11 Definition of *labelRule* to append a *Rule*'s identifier. 49
4.12 Defining the strategy in a phase of a conversation using interleaving. 50
4.13 Defining a strategy in phases. 50
4.14 Defining the strategy in a tree like manner. 51
4.15 Code of an example strategy, in action–reaction pairs. 52
4.16 An example strategy of interleaved, agent–specific sub–strategies. 52
4.17 A first definition of rules and strategy. 53
4.18 Specific constructors for *Interaction*s allow to differentiate interactions on the intention of a sentence. 53
4.19 An illustration of an easily made mistake, when using *String*s as keys, in a GHCI session. 54
4.20 Example of a *Map* with type–checked key values. 54

5.1 Context–specific definitions of two similar interaction. 56
5.2 Defining a *Context* for client information enables the context–dependent definition of interactions. 57
5.3 Example modifier for making contact by addressing the client by name. 58

Preface

After finishing the courses for the Computing Science master at Utrecht University, I had difficulty deciding on the subject of my research subject. Luckily after a few talks with Doaitse Swierstra, Atze Dijkstra and Johan Jeuring, I found a subject that suited me. At that moment I was thrilled to start my master research! During my project I've learned that I truly am a programmer. I like to understand problems, think in solutions and let a computer solve problems using my algorithms. This project not only challenged me to implement a piece of software, i analysed a real–world situation and transfer that to a programming challenge. I worked together with professors of Utrecht University, but I was on my own to plan and manage a prolonged project. Another thing I noticed were the pitfalls of doing research alone. Let's say there have been more and less productive periods, and that I have learned that a good planning is key to progress. Now, the project is finished and I'm happy with the results.

First of all, I would like to thank Johan Jeuring and Bastiaan Heeren for the support and many talks about the IDEAS framework. Secondly, I thank Majanne Wolters for discussing the training and developing the example scenarios that formed the basis for the exercises. I would like to thank the members of "Communicate!" and students in the project group "Sprout". They kept me motivated by working towards a shared goal, the communication game. Thanks to study–association Sticky for lots of coffee, an occasional beer and mental support. And lastly, my parents and friends supporting me and listening to my day–to–day struggles.

I want to thank everyone for their time and effort during my Master research project.

Frank.

Chapter 1

Introduction

1.1 Motivation

The motivation for this thesis originates from the need for research on developing a serious game which assists and trains students to communicate with a patient or client. Utrecht University started a project called 'Communicate!', for developing a game in 2013 and following years. Students and teachers work together to develop a communication game.

The investment of universities in serious games is not unexpected. Stapleton [1] identified the higher education sector to offer the greatest potential for the development and implementation of serious games. He argues that academics are likely to contribute to development and have the capability to build products for themselves. Another reason that Utrecht University invests in digital learning environments is financial. The current recession encourages faculties to explore new ways of education. Cutting on the costs of education can often be achieved by an IT–solution.

Doctors, psychologists and pharmacists rely heavily on communication with patients or clients. For this reason, many of the bachelor and master programs train their students soft skills. Students train their communication by doing time–consuming exercises. A student from pharmacy for example, has conversations with other students or actors. Given a computer game that simulates a conversation partner, students can practice more efficiently. A serious game can save money spent on actors.

The goals of most serious games are to facilitate gamers learning higher order thinking skills.[2] The 'Communicate!' project aims to develop computer game that allows a player to train communication skills. Susi et al[3] have investigated the use of serious games and expect that the education and training market will see an increase in the use of serious games.

Learning by playing a game has some advantages over alternative ways of education. Serious gaming allows learners to experience situations that are expensive in money or time, in real–life. Secondly, some skills can only be taught by doing, but putting a student in a real situation can be risky or harmful to the student and bystanders. Utilizing games for learning provides a safe situation at a lower price. Lastly, the flexibility of education increases. Digital tools allow a student to do exercises at any place and time.

1.2 Goals

This research is aimed at the analysis of communication training and implementation of a framework for a serious game. We want to simulate communicating with human–like actors without the real–life limitations.

We think a serious game depends on the software that generates exercises and feedback. Developing a serious game, like any piece of software, is usually a process of incremental refinement. Some design choices are best investigated before starting development. In this thesis we compare alternatives to generating exercises for a communication game.

Feedback is an important factor for learning because it shows how good the learner did, and points out how the learner can improve. Our system gives feedback, such that it resembles the communication training, and that it can be calculated efficiently in the data model. According to Erev et al[4] different kinds of feedback lead to different results. Feedback is defined by the input from a teacher. The teacher should not be restricted to choose from the possible ways of feedback.

Heeren et al. have created a framework for giving feedback on specific domains. [5] [6] The framework was initially developed for serving and generating feedback on mathematical exercises. Later on, domains like logical formulae [7] and a functional programming tutor, Ask-Elle[8] were introduced. The framework generates feedback on how a student is solving an exercise. IDEAS checks a student's actions, proposes rules and recognizes a student's error. Exercises adhere to the abstract structure that consist of a problem and rules that manipulate the problem. For example, a mathematical formulae and rewrite rules. Communication might seem a different problem from solving a mathematical equation and part of this research is to find the problem and rewrite rules in the domain of communication training.

In this thesis, we aim to analyse communication training and compare possible solutions to support a serious game. This thesis works towards an implementation of a framework, in which we can employ different scenarios for training communicative skills, and which allows to automatically analyse and give feedback on the student's actions. Accordingly, we will answer the following questions:

- What are the identifiable concepts of training communication?
- What are the functional requirements for simulating a human–like actor for communication training?
- What requirements can we use to compare software frameworks?
- What modelling paradigm should we use to base our framework on?
- What do we have to change to an existing framework to allow its use for communication training?
- What problems and choices do we encounter when designing a framework for a serious communications game?

We divide these questions in three chapters. We start by analysing the conversations at the communication training of Pharmacy at Utrecht University.

Chapter 2, featuring the training analysis, answers questions one and two. Secondly, we compare the different frameworks that could be selected, based on the functional requirements. We give the requirements for a software framework for simulating conversations similar to training. We answer the third till fifth questions in chapter 3. In chapter 4, we implement a specially developed scenario in the IDEAS framework. Moreover, we discuss the choices for implementing such a scenario. In chapter 5 we conclude and discuss future work and related ideas.

Chapter 2

Analysing Pharmacy's Communication Training

In this chapter we look at how the learner is currently taught how to communicate. I attended communication training sessions that are part of a Pharmaceutical Sciences master course at Utrecht University. The sessions are focused on the conversation between a pharmacist and a client that fetches his medication. Students learn how to talk to clients about the effects, side effects and proper use of their medicine. An good conversation with a client improves the effectiveness of the use of medication. The better a client is informed, the higher the chance that no complications arise and the medication works as it was supposed to. I sat with the students during their training to analyse the specific concepts of communication for a pharmacist.

The training is split up in three different sessions. In the first and second sessions, the theory of communicating with a client is discussed. Those sessions introduce real–life conversations with clients, common to a pharmacist. In all sessions, students spend some time practising conversation based on cases that stay close to real conversations behind the counter. The third session revolves around how to deal with difficult clients. Difficult clients are clients that don't want to cooperate with the pharmacist. A professional actor is hired to play the role of difficult clients in the third session. From now on, whenever we mention an actor, we mean either a student playing a client or a professional actor. When we talk about the student, we mean the student that plays the pharmacist's role. Every conversation ends with a round of discussion to give feedback.

Interesting for this study is how we can translate the communication training to exercises for a serious game. Before we can define exercises, we need to identify every aspects of training in the course. Next we look for the parts of the training that we want to offer digitally to the students. We then set requirements for a framework which accommodates that part of the training.

While breaking down what we want to analyse, we set ourselves a couple of questions:

1. How is the theory presented to the student?

2. What information is input for students to practice conversation?

3. How do students and the teacher give feedback to the actors in the conversation?
4. Which concepts can we identify that should be defined in a software framework?

The following chapter is structured as follows. First, in section 2.1 we look at the presentation of theory. Section 2.2 explains how students practice communicating. In section 2.3 we explain how feedback is given. We list what a student should be able to do after completing the training in section 2.4. Concluding in section 2.5, where we explain the requirements that we could identify.

2.1 Theory

A pharmacist communicates with clients about the client's situation and medication. The client comes to the pharmacy with a specific goal. These goals differ but can be sorted in three groups. Within a group, a pharmacist has a specific role. Blom et al.[9] divide conversations in a pharmacy in three groups, namely:

1. First prescription contact. A client's goal is to fetch medication which the doctor has specified as part of the treatment. The client might not be known with the medication. During the conversation the pharmacist explains how to use the medication correctly. Problems might show up when the proper use of medication is explained. For example taking eye drops for a client with Parkinson's disease, or drowsiness for a taxi driver.
2. Second prescription contact. In this case, the client has used medication once before and has come to fetch more medication. The pharmacist checks how the client experienced the use of medicine. The client can have a range of goals, from completing the treatment, to stopping it. Problems can arise, and have to be dealt with.
3. Self–care. Clients can come without prescription of a general practitioner. The client's goal is to explain the symptoms and raise concerns to the pharmacist, often to see if there's a medication that could relieve the symptoms. The role of the pharmacist is to identify whether the pharmacist can diagnose the client or has to refer the client to his general practitioner. If applicable, the pharmacist can suggest drugs that alleviate symptoms.

Part of the training is to learn how to communicate with the purpose of the pharmacist in mind. Identifying what group a conversation with a client is in, is the first step for the pharmacist. The first question of the pharmacist is often: "How can I help you?". From that point, the structure of the conversation is typical to the goal of the client. Most students easily identify for which of the three goals the client has come to the pharmacy. The difficult part for students is often keeping a conversation structured and concise without interrupting a client. To keep track of the group–specific role of the pharmacist, the training presents models. Models give the student a high level description of a conversation.

The training doesn't merely cover the high level structure of a conversation. It also teaches how low–level communication skills aid the pharmacist. Skills

are the tools that the pharmacist uses to communicate effectively. For example, making contact with the client ensures that the client is interested to listen. When the client listens to and focusses on what the pharmacist has to say, the instructions on using medication and information about (side–) effects are received better by the client.

2.1.1 Models

Earlier we identified three different groups of clients with similar goals. The students learn an set of models specific for each of the three kinds of conversations. A model structures a conversation by defining phases.

For example, in the self–care conversations students use the “WHAM” model. The use of “WHAM” is described in “Standaarden voor Zelfzorg”[10], which loosely translates to “Standards for Self–Care”. It is published by the KNMP, which is a Dutch organisation that represents the interests of pharmacists. “WHAM” stands for four questions that need to be answered, to asses what a pharmacist’s next actions should be. In some cases, a client can be treated by the pharmacist. A pharmacist can advise on self-care products. A pharmacist sometimes will refer a client to the general practitioner when necessary. Translated from Dutch, “WHAM” stands for:

- Who is the advice for?
- For how long has that person had the symptoms?
- What has that person tried to alleviate the problem?
- Does the person use any medication?

Every conversation starts with an introduction phase to identify the client. The pharmacist concludes the conversation with summing up the important aspects and asks if the client has questions..

The model for self–care is a mnemonic that sums up the four phases for that specific type of conversation. With this mnemonic, a pharmacist can check if every phase of the is handled correctly. Sometimes a client influences the conversation and skips a phase. Even though the pharmacist loses control over the order of the phases, the model offers a way of keeping in mind which phases have been discussed. If used correctly, the models bring structure to the conversation and guide a pharmacist to follow the preferred order of phases. At any given moment in a conversation, the pharmacist can think of mnemonic to remember what has been, or what should be discussed.

2.1.2 Skills

Students are taught basic skills to conduct effective communication. A client and pharmacist should both get the opportunity to speak and should listen to each other. That is the only way to share information and prevent problems.

For example, making contact should always be at the start of a conversation. There are a couple of ways to make contact and one is not always better than the other. Relating to the conversation partner is key to a good conversation.

When the pharmacist has established contact with the client, the models come into play. As he works through each phase of the model, he should

take pauses to reconnect. The pharmacist cannot assume that the client stays focussed while he is talking through the effect, use and possible side–effects of a drug. It is good to keep the attention of the client to ensure the client is focussing on the conversation.

The pharmacist needs to understand the client to give personal advice on the use of medication, for example. To understand the client and its situation, the pharmacist can ask questions. Sometimes it is good to ask a directed question, but often an open question is the better option. Students are taught that they should pick the words they personally prefer, but that some questions won't find the answer they are looking for.

The use of open questions is easily illustrated. Closed questions assume certain shared knowledge, which is often not the case. In Table 2.1, you see that the pharmacist assumes the client knows the correct way of using medication.

Pharmacist:	Did you use the medication as prescribed?
Client:	Yes, I did.
Pharmacist:	Did you use it twice a day?
Client:	Yes, i took two pills each day
Pharmacist:	Did you use one in the morning and one in the evening?
Client:	Ohh, no, i took both in the evening. Is that a problem?
Pharmacist:	Well, perhaps that is the cause of the side effects.

Table 2.1: Finding out how a client has used their medication. An example of closed questions in a conversation.

Luckily, they find out shortly after that this assumption was not justified. It is only by the follow-up question that they find out that the pills were not taken correctly. Although the pharmacist found out how the client has taken the medication, there is danger in assuming that the client remembered how to take medication. An alternative opening question would have been: "*How did you take the medication?*"

The right questions and careful listening to a client reveals the experiences of the client. It is important to understand the client. Understanding the client is a must for giving advice tailored to the situation and needs of the client. Two other skills presented are paraphrasing and summarizing. These two give the client the idea that he is heard by the pharmacist. When used correctly, good questions, listening, paraphrasing and summarizing create mutual understanding.

Mutual understanding of the situation allows problems to be found and discussed. When there is discussion between the pharmacist, who knows all about the medication's use, effect and possible side–effect, and the client, who needs to find a way to use the medication correctly in his daily life, the best working solution is found. Also problems with the clients lifestyle and use of medication can be found, before the treatment is started. For example, a drug that makes the user sleepy is not advised for a airline pilot. Giving and taking advice is more effective when the client and pharmacist have discussed the situation.

Furthermore the students are pointed to cues. Cues are (non-) verbal signals that can show the concerns and emotions of a client. For example they often

show if the client is paying attention, listening, and trying to actively take part in the conversation, or that the client is distracted. Students are advised to address cues by naming what they see, and ask the client to explain. Since cues are often subject to interpretation, it is important not to assume, but to verify with the client. All these skills are tools in conversation to efficiently inform clients how they should use their medication.

2.2 Practice

After the theory has been discussed the students form pairs. The students either play the pharmacist or client roles in a counter conversation. The specific details are described in a case. The pharmacist takes place behind a counter. The client enters the pharmacy and walks towards the counter, and the conversation begins. While the client's main goal is to receive the medicine and take it home, the pharmacist wants to be sure that the client knows how to use the medicine and increase the probability that the client uses the medication.

Paired students play a case twice, allowing both students to practice as the pharmacist. After each play the students are encouraged to discus how the pharmacist fulfilled its role. The students use the theory to substantiate peer–reviews for each other. Even though some students don't play the role of client very realistically, the two students together often find points of improvement for the student in the role of the pharmacist.

Being able to handle any client in a conversation is the goal of the training. The course is completed when the students can identify problems and inform the client in a test. During the test, the client is played by a professional actor which ensures a realistic setting and a constant level of quality. Both the feedback in discussion and scoring on the practical test are based on the theory.

2.2.1 Cases

The cases are the starting point for the conversation, and hold information for a client to play its role. A case supplies information about these three aspects of the conversation: context, client and scoring. Cases used for the test often are variation of other cases. A different context or client can make a big difference, and require for different questions in the conversation.

The initial subject of conversation is often the proper use of medication. The context for the training is the specifics of some drug and the description of the client's lifestyle. The student's knowledge about the domain is assumed to be sufficient and is not tested by the final exam of the course.

Figure 2.1 shows an example case description that is given to the students to train with. First the medication is specified and it states the dose for this client. Secondly the name and address of the client is given. And lastly, the situation of the client is explained in a few items. These last pointers on the client's situation make the case interesting for students to train.

Domain specific information gives the conversation a sense of realism. A conversation that is not based on a realistic subject is hard to play out. In the pharmaceutical domain, the specific characteristics of medication define the problems that clients could face. Furthermore, specific details are given to

Alendronic acid 70mg, 12 pcs
S 1 x per week 1

Second prescription - case 11 round b

J. Pietersen
Stadhuisplein 70

Situation of the client:

1. Took second capsule a day later, is that bad?
2. Struggles to remember to take the medicine.

Figure 2.1: Example of an case description card, used by a student to play the client's role.

explain more of the problem at hand. For example, that the client works as a pilot or that a woman on birth control has had sexual intercourse.

Even though the dialogue knows two active actors, other actors can be of influence to the current situation. A client's general practitioner and the pharmacist's colleagues might be part of the context. For example, the general practitioner might have informed the client in some way, and indirectly is contributing to the conversation. Another example, in a second prescription contact setting, a pharmacist might be accused for something that a colleague did. And lastly, in self–care conversation for example, a mother can come to the pharmacy to get medication and advice on how to help her son get rid of pin–worms, and prevent her family getting infected by her son.

The client is described such that the student or actor playing the role of the client can do so realistically. The case defines the character of a client, specific behaviour and specific conversational directions.

The character of a client allows the actor to choose specific wordings and to give subtle cues. A character description gives a background for the conversation. For example, the actor might look away often, which shows that the client is not focussed.

A case description might state that a patient desires a short conversation. An example would be that when the client is waiting, he gets agitated. If the conversation takes too long, he might cut off the conversation and leave the pharmacy.

Sometimes a client has specific concerns. The case can describe specific topics about which the client has questions. Sometimes the case gives directions to steer the conversation a particular direction. A client's reaction can give a clue to the pharmacist to investigate further. To guide the conversation during the test, answers to the pharmacists possible questions are noted for the actor to use.

The case gives pointers on scoring the performance of a pharmacist. Goals are stated such that feedback can be based on them. A goal could be: find out that the client is a pilot, and that he should not use medication that has sleepiness as a common side–effect.

Furthermore pointers for feedback that are specific for this case are given. A case exposes a set of conversational aspects. For example from the examiners perspective, scoring a pharmacist for interrupting a client that is not talkative seems difficult. In this example a pointer could be to make contact in such a way that the client trusts the pharmacist and is willing to speak freely.

2.2.2 Conversation

The practice during training sessions can be described as conversations between two agents. The students only play clients because the pharmacists needs someone to speak with. A conversation has one active agent and a simulated agent.

There are some drawbacks from students practising together. Even though there is no prior knowledge for the pharmacist, students often react as if they feel it is not real. It is of course a play, but these students know each other better as fellow students. Some students find it hard to play the roles convincingly, and break character easily. For example, in a case about a woman using contraception medication, students find out that there is a situation where the woman might be pregnant due to incorrect use. An unwanted pregnancy is of course an extreme case. Few students can act professionally when such extreme situation occurs. Sometimes, another effect of students practising on each other is immediate feedback. When the pharmacist makes a mistake, some clients break character and give directions. A client tries to address a subject again because the pharmacist did not catch the problem at hand. For example, the pharmacist asks if the client has used the medicine correctly and the client confirms. The underlying problem is that the client does not know that he uses it incorrectly. The pharmacist might think that client uses it correctly, but in fact the client didn't.

The client resembles an intelligent agent. Simulating the client could be achieved by defining an intelligent agent. Woolridge[11] defines a weak and strong notion for agents in artificial intelligence. The weak definition would describe requirements for simulation of a communicating agent. The participants of communication would at least have the four properties of agents: autonomy, social ability, reactivity and pro-activeness. A simulated agent is able to select its interaction, by either reacting to another agent or computing it pro-actively.

Even though the students have all freedom over what they say, the practice conversations are only a couple of minutes, which means the number of interactions is limited. Even with a client that is very talkative the conversation shouldn't go on to long. The goal of a conversation with a client who for example, excessively makes small talk, is to interrupt in a good fashion, such that the contact between client and pharmacist stays intact.

Being straight to the point and giving concise advice is a must for pharmacists. A duo gets five minutes to practice a case and a conversation ends when the time is up. It is a goal to end a conversation within these five minutes. Students often experience that they have too little time to finish the conversation.

Due to the time limit a conversation sometimes ends abruptly. Students who try to focus on the time limit, sometimes don't find the problems of the client. In the example of the pregnant woman, the pharmacist advised the client to proceed with the medication. What she didn't know was that her advice could

have caused serious issues to the unborn child. That some conversations end unnaturally doesn't mean further discussion is less of value.

Interactions

We define an interaction as the actions in a conversation. Interactions consist of utterances, but they also convey non-verbal communication. Interactions affect the conversation and the actors of a conversation. The effect on actors can differ due to the interpretation of a utterance or non-verbal act. Another effect of using an interaction is that the conversations proceeds. And lastly, interactions are not always applicable, there is a time for every instance of an interaction.

Van den Bosch et al.[12] distinguish three types of interaction: Tell, Ask and Acknowledge. Ask and acknowledge are equal to question and answer, respectively. In our definition, we distinguish two ways of telling something. Answering to a question is different from outing a concern or informing someone. Also, they do not distinguish interrupting someone as a type of communication.

When looking merely at a verbal interaction we distinguish varying goals. We can define different five types of interactions, namely: questions, answers, concerns, informs and interruptions.

1. Question. When an participant finds a certain variable interesting it is polled. For example, the pharmacists asks if there were any problems with taking the medication.

2. Answer. After a question, an answer is expected, and naturally the pharmacist will wait for the client to answer. Questions and answers are connected to each other.

3. Concern. A concerns show how a client thinks about his situation. A client wants to let the pharmacist know what is troubling him. Raising a concern different from reacting to a question. A concern is the desire of the client to state something.

4. Inform. Similarly as a concern, an informing interaction does not answer to a question. The pharmacist wants to explain something even though the client did not ask a question.

5. Interruption. In communication, there is a flow of interactions. Interrupting a speaker can break this flow, and allows another participant to become speaker. In natural speech interruptions are needed if there is not set pace for listening and speaking. When there is moment in which you get the option to speak, interruptions are not necessary.

Furthermore Van den Bosch et al. developed building blocks to describe interactions. They developed: fact, interpretation, opinion, wish, importance, argumentation and illustration to define dialogue. These blocks are used to label interactions. Building blocks together with the types of interaction, describe an interaction such that a computer can use it.

Using building blocks limits the writer to define interaction in his own words. We can use strings to label interaction with extra information, allowing us to give specific names to interactions. We refrain from using predefined building blocks to define interaction. Labelled sentences better convey what an interaction means.

2.3 Discussion

In the discussion after playing a case, the students give feedback for the pharmacist. The two subjects of feedback are the client's situation, and how the pharmacist did.

Often students discussed the validity of their advice. They are wondering if they were a good pharmacist in the conversation. Many first questions for example, were similar to: "Was my advice the correct one?". Other questions on the clients situation where about the correct use of the medication, or alternative medications to solve certain problems. In the example of the drug–induced sleepy pilot, a simple solution was to give alternative medication. These questions are not about communication, but about pharmaceutical knowledge.

Even though the subject of the training is to teach the students to communicate, they are focussed on their field of study. It is unclear if feedback should be given on the pharmaceutical knowledge. Students might desire that a conversation is correct according to the pharmaceutical context. On the other hand, the training focusses more on domain specific communication then on a student's knowledge of the pharmaceutical domain.

Luckily students often recognize communicational problems in a conversation. Because both the actors in the play are students, they both can reflect on performance of the pharmacist. The mistakes they identified are often the subject of the discussion. For example, a student did not find out that the woman had sexual intercourse while not taking birth control pills, because he lets the client talk about the weather. These mistakes are simple, but more hidden problematic situations are found when relating to the theory. The problem of the pilot was not found because the pharmacist did not ask if the client understood how the medication is to be used. When asked, the pilot might have said something like: "The problem is that I cannot be sleepy while working." Which would make it easy for the pharmacist to follow through with a solution.

2.4 Goals of Communication Training

The communication training course concludes with an exam, where an actor plays the role of a client. Students perform similar conversations to those of the training. The actor is instructed to not break character, and is not allowed to give hints to the student. Moreover, the final exam tests the student's capability to have an effective conversation by meeting the goals. During the exam, teachers score the student on five items.

- Understanding the client. By listening and having an open, inviting attitude, the client can speak freely. The student must be capable of analysing a question and decide how to address the situation.
- Informing clearly. The student knows how to inform a patient unambiguously and in portions that the client can understand. The student checks if a client has understood the information.
- Patient alignment. The student makes contact to the client, such that the client is approachable and willing to listen. The student can reach the solution in the clients specific situation. The style of communicating fits with the client's needs.

- Professionalism. The student can take the lead in the conversation without upsetting the client. Furthermore, the student takes responsibility for the conversation, but respects the autonomy and personal responsibility of the client.
- Pharmaceutical therapy. The student apply the conversation models, gives the essential information on characteristics of the medication. In the case that a client is in a hurry, the student is capable of selecting the important aspects to tell to the client. A student must be able to select a correct treatment, in the case that a client comes for help without a prescription. Lastly, a student should be able to incorporate the use of medication and side–effects to identify, and seek for solutions to medicine related problems.

Schaafstal [13] presents three layers for diagnosing skill. The three layers concern fulfilled tasks during execution, the knowledge of relevant local strategies and the underlying domain knowledge. The communication training covers both first and second layers. The tasks that students fulfil, relate to passing the phases of a model. The class would discuss whether a student reached all the goals accompanying the various phases, or that a student should have been more thorough. For example, when a student should have asked more about experienced side–effects. Local strategies relate to the ways of directing communication and structuring a conversation correctly. When the conversation deviates from the subject, the pharmacist should direct the conversation without being rude to the client.

In gaming, some learning goals are more easily assessed than others. For example, it is possible to assess a students professionalism throughout the conversation. We can define which interactions are generally unprofessional, and accompany that interaction with a negative score on professionalism. More difficult it might be to assess if a student is listening such that a client feels invited to speak freely. Listening is more of a passive attitude than a activity. What we can assess, is whether the actions that follow after a clients interaction show that the student has been listening and understands the client. We think that a large portion of the learning goals can be trained and assessed in a serious game.

2.5 Requirements for Defining Exercises

The translation from the training to a simulation require the definition of exercises, which come with a couple of requirements. To be able to define exercises similar to the practice during training, we incorporate the information from the case description and medicine. A client can be simulated by defining answers to questions and concerns. Another possibility would be to define agents that can react intelligently based on the information of the client's case.

The conversation can be built up by interactions. We require the following information concerning a single interaction:

1. Utterance. We need a way to show interactions to the student. An interaction's utterance can be text or audio–clip. Perhaps accompanied by an piece of film or human avatar to create an natural environment.

2. Time. If we have a conversation of a couple of interactions, we need to know the ordering between them. We want to know which interaction is valid at what time. This could imply a higher structure that defines when an interaction is applicable.

3. Effect. Lastly, an interaction has (implicit) effect such that the conversation will progresses further. The effect should be noticeable for the participating agents, such that a next round of simulation results in new interaction, and a student would know how to react to the new situation.

We will want to describe how a client reacts to questions, and raise its concerns. The training simulation will hold information about the pharmacist and client's possible actions. Using these requirements, we can select models to implement a scenario for training a pharmacy student.

Chapter 3

Comparing Frameworks for Communication

Many developers of games with non playing characters encounter the same problem of modelling conversation at some point in their development. The goal is to create non-playing characters (NPC's) that are interesting to talk with and give realistic reactions to questions. To have a realistic conversation with an NPC, he is required to react coherently and only utter sentences that add to the conversation. Schwarz et al. [14] call these requirements the Rules of the Communication Game. Many games feature conversations, but interacting with NPC's is not often the core of a game. In a game where conversations are a side feature, it is desired that defining NPC's requires little modelling and are controllable in the sense that they say what you want them to say. We have found three ways of modelling conversation and analyse the pros and cons of each approach in the context of an educational game.

Section 3.1 explains the use of Haskell for code examples to follow. We set the requirements for frameworks in section 3.2. Next, section 3.3 walks through the example scenario. The sections 3.4, 3.5, 3.6 respectively discuss frameworks using dialog trees, BDI models and domain reasoners in more detail. We conclude our comparison in section 3.7

3.1 Preliminaries

To be able to make a comparison for software frameworks, we define them and analyse them on a set of requirements. Comparing implementations require that some variable aspects are set for all frameworks. We pick a single programming language to implement the software in. Although arguable that some frameworks might be easier to implement in some specific language. Using a language that is specifically good in defining some kind of framework does not adhere to the requirement that a framework should be easily readable. Secondly, comparing implementations on their size would make less sense, if the programming languages differ.

Another variable is the example exercise that we want to describe as a test–case. We use a single example conversation for implementing every framework. Every implementation describes the same scenario such that it does not affect

the comparison of the frameworks. We take a part of a scenario that was defined by a teacher of the Pharmaceutical sciences master course as an example of an exercise. It is a realistic prototype case that is written with education in mind. The current example scenario uses multiple choice options for the pharmacist. Our implementation does allow for alternative forms of assessment. Scalise et al[15] identified ways of assessment. They define seven groups ranging from true/false questions to giving a presentation. We could implement the two most constrained types: multiple choice and selection/identification.

3.1.1 Haskell

To test alternative frameworks, we have programmed simple proof–of–concept programs that show the implementation of a framework, and the definition of an exercise in that framework. For these programs the language of choice is Haskell. It is an advanced, general-purpose functional programming language. It has the advantage that it can create concise software, which helps to show the concept to the reader in code rather than its documentation. Furthermore functional languages in general allow the programmer to create correct, type–safe software due to the pure nature of functions.

Figures in the following sections contain source code. These examples use conventional line numbering, which is reset for every section. Line numbering should make it easy to differentiate the definition of one framework from the other.

In the following sections, some code is excluded for brevity. Sometimes type signatures are included to allow the reader to understand what the semantics of a function is.

3.2 Technical Requirements for Software Frameworks

As a result of the analysis of the training communication we describe our conversation as interactions between two virtual agents. One agent is our player or, from an educational point, a learner. The player decides the actions of the virtual agent for the pharmacist. The second agent is a non–playing character (NPC). The computer decides or calculates the NPC's actions.

Software quality is a well–debated issue.[16][17] The international organisation for standardization has a specific software quality standard named ISO/IEC 9126. The latest addition is the standard for defining a software quality, ISO/IEC 25030[18], which takes stakeholders into account. For comparing frameworks it is at least required to have one or more measures from a quality model. The following sections explain the concept which we use to compare the different frameworks.

3.2.1 Control

To describe a simple conversation between our agents we need to identify the actions and reactions of agents. In an educational game setting, a conversation follows a scenario. That scenario is written by a scenario writer, such that it allows the player to learn. For example, we want that a student could repeat

an exercise with the same reactions by the NPC. The NPC is required to act unambiguous in multiple runs of an exercise. Having control over the NPC and its actions is a technical requirement of a framework.[19]

3.2.2 Usability

Another technical requirement of software in general, is usability of the source code. It concerns the ability to operate and learn about the framework. The user of a software framework should be able to read the code, and understand what it does without intensive study. Especially for a small system for conversation the focus lies in creating a usable framework, rather then a framework that can deal with every possible situation.

3.2.3 Maintainability and Adaptability

A framework should be maintainable and adaptable. In software quality maintainable means we can analyse the framework and proceed in correcting errors, thus verifying the correct functionality. Adaptability is an important requirement since software in development often needs to change to new requirements. For example, when the project is extended to other fields of conversation training. If we want the framework to be adaptable to future definitions of conversation, we need to use a maintainable and portable framework.

3.2.4 Generating Feedback

It is important that we can give feedback to the learner at any moment in or after the conversation. During the conversation we collect or build up parameter values that reflect the learners performance. To allow the computer to compute feedback, the interaction chosen by the player modifies one or more parameters. We define parameters and what it means to have a high or low value on that parameter, to reflect how the player has done. For example, we could define a contact–parameter to keep track of the learners ability to make and keep contact throughout the conversation. If an interaction has a high value on the contact–parameter, choosing that interaction means making or keeping contact with the NPC. When scoring high on the contact–parameter, the system can give feedback that the learner made contact with the NPC using a particular interaction. The computer can show how the current option compares to other options. For instance, if there is an earlier interaction in the conversation, which makes contact, the learner will get feedback that it could have made contact earlier. Furthermore, we can use the values on these parameters to create a report. These reports allow comparison of conversations and discussion amongst learners. An exercise that contains many interactions that modify the contact–parameter trains the skills to make or keep contact with the other actor. If a learner has difficulty in making contact, we can suggest an appropriate follow–up task. Suggested tasks are a form of feedback that allows user–customized learning.

3.2.5 Concise Implementation

A framework's well–defined data structure uses concise Haskell data types and functions to be expressive. We can compare the codes of the implementations statically, by counting the number of data types defined that are needed to implement a framework. Some frameworks use helper–functions to build up a scenario in the framework, which can be compared by number and complexity. Furthermore, we can analyse the number of constructors used when defining the example scenario. By comparing these numbers, we compare the frameworks on their structural complexity.

Creating a realistic and interesting conversation is the challenge of the writer of a scenario. The writer provides the proper domain knowledge and context for the conversation. It is preferable that the framework allows a clear and concise definition of interactions to describe a scenario. Furthermore, the changeability of the scenario should be taken in to consideration. An framework that allows changeable scenario descriptions allows incremental development rather than having to start over with development, every time an addition or change is issued. The ease of use and adaptability of a framework is defined by the data structure, and the functions that manipulate the data. Summarizing, we should compare frameworks on: data structure, control over dialogue, amount of work to write a scenario, adaptability of conversation and possibilities to encode feedback parameters.

3.3 Interactions of an Example Scenario

We assume every conversation is built up from interactions and we use these interactions as the starting point for every framework. Before we can define the example conversation, we will have to define the interactions. Every framework will use the same interactions. As seen in Figure 3.1, *Interactions* contain an identifier and a sentence. The sentence can be used in a game to make the interactions readable to the learner. For the sake of simplicity we define the sentence as a *String*. In a serious game other media formats like film or an animated avatar could be added. The identifier is a number in our case, but for larger scenarios we use the identifier to encode more information. We use the identifier to add hierarchical information about a rule. In the identifier, we could specify:

1. The conversation. A conversation with a client without prescription, with a sore throat can be described by `"self-care-cough-syrup"`.

2. The phase of a conversation. For example: `"introduction"`, `"explain-drug"` and `"conclusion"`.

3. An intention. I.e. `"find-problem"`, `"make-contact"` or `"summarize"`.

4. A particular skill that is used. Specific communication skills like `"open-question"` and `"reflect"`.

An interaction does not give enough information to calculate feedback for a learner. We could extend each interaction with modifiers of certain parameters which hold extra information about this specific interaction's effect. To give

```
1 data Interaction = Interaction {
2   identifier :: Int,
3   sentence :: String
4     -- parameterModifier :: SomeDatatype
5 } deriving (Show, Eq)
```

Figure 3.1: Data type definition of interactions.

feedback on the performance of the player, we can use these parameter modifiers to show the actual score.

3.3.1 Example: Miss Darcy Fetches Her Metformin 500mg

The interactions together form a piece of conversation between a pharmacist and a client: Miss Darcy. Miss Darcy comes to get her "metformin 500mg". The conversation starts with the first choice for the pharmacist. Each of these options has an effect on some parameters, which can be excluded for brevity. The first four options are:

1. This is your metformin 500mg.
2. Miss Darcy, here are your drugs.
3. Miss Darcy, what did your general practitioner tell you about metformin?
4. Is it correct that this is the first time you take metformin?

Every option results in a single return statement from the client. The identifiers for the client's reactions are characters. The reactions to the first four options are given the letters 'a' to 'd'. If the pharmacist gives the metformin, using the first interaction, the client reacts with 'a': "Yes, I assume so". The pharmacist next has the option to answer with:

5. Miss Darcy, what did your general practitioner tell you about metformin?
6. Miss Darcy, is it correct that this is the first time you take metformin?
7. Did you know it was called like that?
8. I have the impression that you are absent-minded?

Defining a scenario with four unique sentences every option is infeasible as a scenario is expected to have many interactions. If every option for the pharmacist had four unique interactions, we would need four interactions per option. And for each four actions of the pharmacist, we have a reaction by the client. This results in a total number of specified interactions i of

$$i = \sum_{n=1}^{k} 2 * (4^n)$$

where k equals the amount of choices for the pharmacist. A solution is to reuse interactions in paths where they were not chosen before.

For example, when we choose the first option we are given the interactions 3 and 4 again. As option 5 is equivalent to option 3 and option 6 is similar to option 4. Had the pharmacist chosen the second option, the client reacted with 'b': "Yes, thank you". The pharmacist could proceed with:

9. This is your metformin 500mg.

10. Miss Darcy, what did your general practitioner tell you about metformin?

11. Is it correct that this is the first time you take metformin?

12. I will tell you a bit more about the drug.

In the case of the third option in the first choice, the client would have said 'c': "Well, that I have to use this." Possible reactions of the pharmacist are:

13. Is it correct that this is the first time you take metformin?

14. Yes, that is correct, it is important for the diabetes.

15. Did he tell you how to use it?

16. So the general practitioner told you to use this. Did he also tell you why?

Lastly, if the pharmacist had chosen the fourth option, the client reacts with 'd': "Eh, yes.", and the next options are:

17. I have the impression that you are absent-minded?

18. I will tell you a bit more about the drug.

19. So the general practitioner told you to use this. Did he also tell you why?

20. So you haven't used this before. Do you know why you are going to use it?

Using the definition in figure 3.1, we define the interactions for the short example scenario above as below. Figure 3.2 shows how the four possible opening interactions of the pharmacist are encoded. The rest of the interactions are excluded for brevity, but are defined in similar fashion. With the complete set of interactions, we can start defining the conversation in the following sections.

Using the definition in figure 3.1, we define the interactions for the short example scenario above as below. Figure 3.2 shows how the four possible opening interactions of the pharmacist are encoded.

3.4 Dialog trees

In games, a dialog tree is often used to allow a player to direct the conversation he is in[20]. The player reads dialogue and chooses their response from a limited set of choices available to them[21]. Such conversation is scripted, like for a play but with the difference that the player can choose on certain points what to say. The player has control over the conversation, although the player is limited to

```
 6 yourmetformin      =
 7   (Interaction 1
 8     "This is your metformin 500mg.")
 9 yourdrugs          =
10   (Interaction 2
11     "Miss Darcy, here are your drugs.")
12 practitionertellyou =
13   (Interaction 3
14     ("Miss Darcy, what did your general "
15       ++ "practitioner tell you about metformin?"))
16 correctfirsttime   =
17   (Interaction 4
18     ("Is it correct that this "
19       ++ " is the first time you take metformin?"))
```

Figure 3.2: The first four options in the example scenario, defined using the *Interaction* constructor.

the predefined questions. A dialog tree is a branching description of questions and answers.

We start by defining the data types to create a tree in Section 3.4.1. Section 3.4.2 defines the starting point of a scenario. In Section 3.4.3 we define the first edges descending from the root. We define the branches in Section 3.4.4. In Section 3.4.5 we conclude with an overview of the framework.

3.4.1 Trees

Figure 3.3 shows a visualization of a dialog tree. At the root, the first option for the player, we can choose a direction in the tree. In the example we selected the third option. Each of the four options has an interaction attached to it. The edge that leads towards a new option, has an interaction by the NPC attached to it.

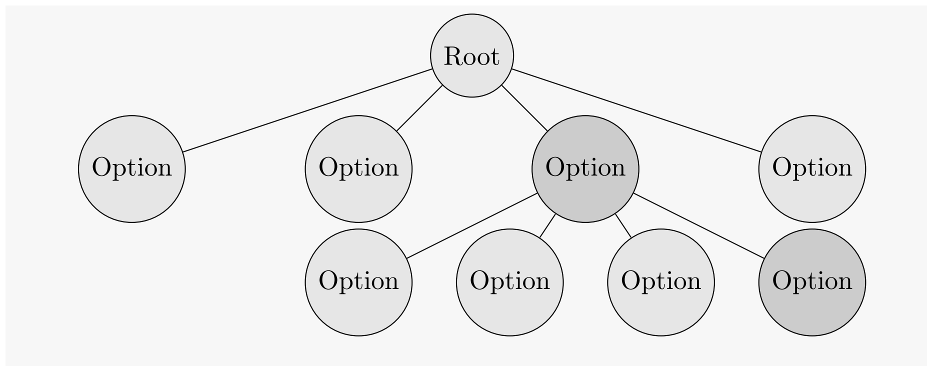

Figure 3.3: The structure of a dialog tree.

A tree consists of nodes which are connected by edges. Every node encodes a (re)action by the NPC and every edge is an step, taken by the player. After

choosing an interaction, coupled to an edge in the tree, the NPC reacts with the interaction as scripted in the next node. Dialog trees allow the player to make decisions in a conversation and can lead to different outcomes. The dialog tree has low computational cost, because at every point in the conversation the player only has the option of the outgoing edges, and the reaction is directly connected. As you can see in figure 3.4, the information is stored in nodes and edges. The exception is the *Leaf* constructor of *Node*, which allows the scenario to end in a reaction by the non-playing character. The actions of a player are encoded in edges, and the reaction by the non playing character is encoded in the nodes. When we encounter a *Branch* while traversing the tree, we can continue with the conversation following the edges of that branch. While a *Leaf* denotes the end of the current path and thus ends the conversation. Since our patient reacts unambiguously to our pharmacists interaction, the *Edge* constructor only links to a single *Node*. Our player needs to make a choice out of the list of edges in a *Branch*. The linked nodes and edges together form a tree, using the *Root* constructor. For simplicity, every conversation starts with a first action by the pharmacist. The root of our tree is a set of options for the player. Finally we can define a *DialogTree* type to be a tree of interactions for both the player and non-playing characters.

```
1 data Edge a b     = Edge b (Node a b) deriving Show
2 data Node a b     = Branch a [Edge a b]
3                   | Leaf                deriving Show
4 data Tree a b     = Root [Edge a b]     deriving Show
5 type DialogTree = Tree Interaction Interaction
```

Figure 3.4: The core data types for describing conversation using a *DialogTree*.

A dialog tree's scenario is fully scripted by the editor. The editor has full control over the reactions by both the player and the NPC. Control over the NPC's personality is implicit, we merely describe the sequence of statements of the agents.

To describe a conversation in a dialog tree, an editor has to give all the nodes and edges. If you want to offer the player a real sense of choice, every node should give multiple options, which would lead to an enormous number of nodes and edges. Rosenfeld[22] stated that maintaining a dialog tree is difficult because it may require transforming the entire tree as new interactions are incorporated. Because each node connects to the following, the addition of a new set of interaction to an existing tree, requires all connections involved in that section of the tree to be changed. If we were to update a single *Edge* in a choice between four, we change the parent–node to add the new and remove the old *Edge*. Subsequently, the new Edge is given the link pointing to the child–node.

3.4.2 Root

The root of the dialog tree is the unique starting point of the tree. Optionally a conversation has a short introduction but we omit the introduction for brevity.

The starting point of our dialog tree is the first set of options for the player. In Figure 3.5 we give a player the choice between four options.

```
6 scenario :: DialogTree
7 scenario = Root startingOptions
8   where startingOptions = [yourmetformin, yourdrugs,
9                            practitionertellyou, correctfirsttime]
```

Figure 3.5: The first option for a pharmacist is encoded in the root of the tree.

From the definition of the scenario, we don't get a lot of information about the rest of the three. To know how broad or deep the tree is, we have to analyse it in a tree traversal. A *DialogTree* is not very insightful from its definition, thus lacking readability.

3.4.3 Edges

These four options are edges that make use of the interactions that we defined earlier. For brevity, the import of the *Interaction* module is qualified as *I*. An edge is nothing more than an *Interaction* and a pointer to the child–*Node*. This translates to the action of the pharmacist and the following reaction of the patient. In Figure 3.6 we see where each of the four options would lead the conversation.

```
10 yourmetformin       = Edge (I.yourmetformin) iassume
11 yourdrugs           = Edge (I.yourdrugs) yesthanks
12 practitionertellyou = Edge (I.practitionertellyou) usemedication
13 correctfirsttime    = Edge (I.correctfirsttime) yes
```

Figure 3.6: Encoding of the four options of the first choice for the pharmacist.

The edges in the first choice each link to one *Branch* or *Leaf* node. Branches would encode an option for the pharmacist. In our example, the conversation continues, so we define *Branch* nodes for the next interactions. Branch elements encode an interaction and the next set of options from which the player can choose.

3.4.4 Branches

At every turn for the pharmacist, we have a multiple choice of four options. A *Branch* defines such an choice in the tree. A branch is defined by an interaction of the client, combined with the four connected pharmacist interactions. In Figure 3.7 we find the sixteen possible states that the player can reach, after two choices. The large expansion of a dialog tree after only a few interactions is evident.

Although we could reuse edges from the first choice, we have to define the edges in Figure 3.8 to define the whole example. In total we have to define sixteen edges and branches to define two choices in our conversation. Reusing

```
14 iassume      = Branch (I.iassume)
15   [practitionertellyou, personalcorrectfirsttime, didyouknow, absentminded]
16 yesthanks    = Branch (I.yesthanks)
17   [yourmetformin, practitionertellyou, correctfirsttime, abitmore]
18 usemedication = Branch (I.usemedication)
19   [correctfirsttime, fordiabetes, howtouse, practitionertellwhy]
20 yes          = Branch (I.yes)
21   [absentminded, abitmore, practitionertellwhy, havntusedwhy]
```

Figure 3.7: Encoding Branches for the players' second choice.

edges decreases the work for a writer but reusing edges might make it hard for a scenario writer to keep track of edges. Definitions used in some state might be scattered since some are defined for this state, and others are reused.

```
22 personalcorrectfirsttime = Edge (I.personalcorrectfirsttime) yes
23 didyouknow               = Edge (I.didyouknow) Leaf
24 absentminded             = Edge (I.absentminded) Leaf
25 abitmore                 = Edge (I.abitmore) Leaf
26 fordiabetes              = Edge (I.fordiabetes) Leaf
27 howtouse                 = Edge (I.howtouse) Leaf
28 practitionertellwhy      = Edge (I.practitionertellwhy) Leaf
29 havntusedwhy             = Edge (I.havntusedwhy) Leaf
```

Figure 3.8: End–states for our example scenario in a dialog tree

3.4.5 Overview

We can now use this example scenario to give possible options to the player, and allow the computer to react to the player's choice. The conversation is built up from nodes and edges, for the player and the NPC respectively. It is a direct and uncomplicated data structure that allows a reader of a tree definition to follow the conversation. We can link edges and nodes together, and that is all there is to it. The structure restricts the writer to alternate actions by the player, and the NPC's reactions. Still building the tree can be error prone because the writer has to define the edges and nodes manually. The writer of a scenario should avoid cycles. Cycles in human conversation make no sense, it is like both agents forgot a piece of their conversation and use exact sentences again. As a tree gets larger, it is more difficult to detect cycles by hand. Luckily, there are algorithms that traverse a tree for detecting cycles. A depth–first traversal can show the writer cycles while developing an exercise.

A scenario defined using a dialog tree is difficult to adapt because of the datas tructure. Major changes to the scenario are problematic. For example, if we would allow the client to react differently based on some variable, we need to introduce a second option in the definition of edges. Changing the definition of *Edge* results in having to change every usage of *Edge*.

All the interactions in the tree should be reachable from the root. Redefining the flow of interactions requires tree transformations such as replacing a path of interactions and correctly connect the path to the existing tree. Keeping deprecated interactions used in an earlier version of the tree intact, is not an option, because every edge and node in the tree is reachable from the root. A disconnected sub–tree is unreachable from the root. A tree transformation requires edges to child–nodes to be reset. Resetting them by hand is error prone because a variable name is easily misspelled.

Say we were to change the reaction on the fourth option in the first choice. This is where the pharmacist asks "Is it correct that this is the first time you take metformin?" and the patient responds with "Eh, yes". If we were to redefine the interaction for the patient to say "Yes, but I know all about it.", we replace the old interaction by the new. The pharmacist will want to act differently to the patient's more confident reaction. For example, a new choice could be "Can you explain how you would use the metformin?". After the patients reaction change, the set of options for the pharmacist is reviewed to match the new reaction. We have to redefine the *Branch* element, with the corrected set of options. The reaction that we changed is reused elsewhere in our *DialogTree*. On all places where it is reused, we have to check whether the conversation should change too, or stay as it was. We might be able to reuse the adapted interaction, but we might have to keep the old one intact.

Lastly, we need information to give feedback to the learner at the end of the conversation. Outcome measures are values that show how good the goals of a conversation are met. We can define how good every choice scores on the outcome measures, and sum these at the end of a conversation. The tree encodes all possible conversations. We need to keep track of the visited nodes in a tree, to encode the current conversation. Detailed feedback information for the player can be given to choice points or sub–trees. We compute the feedback by keeping track of the visited nodes and collecting their feedback information.

The desire of detailed feedback causes interactions to be less applicable for reuse. When we use an interaction more than once in the conversation, the effect often differs. Similarly, an interaction at two different points in the conversation will often have a different feedback information. At one of those places the interaction is a more preferable option. If we want to have detailed feedback, we are stuck with defining a large tree of unique interactions.

Concluding, defining a conversation in a framework using a *DialogTree* is straight forward. Its data type definition is very readable, it is not complex and allows analysis. A conversation is somewhat maintainable, but problems can arise when interaction reuse is common. There is no real overview of the reuse of interactions. Although we would want to reuse interactions where possible as it minimizes the size of the tree, for maintainability reasons, we would prefer minimal use of interactions at multiple positions. The amount of work is increased when we want to have a high grade of maintainability. Using a *DialogTree* requires to choose between a large amount of work and maintainability, while it scores decently on the other requirements.

3.5 Belief, Desires, and Intentions model

A Belief, Desires, and Intentions model (BDI) is an intelligent agent model that selects the interactions of agents on beliefs, desires and intentions[23]. It is a well–developed model, used in many (multi–) agent–systems. In the setting of a communication game, BDI has been used in a game for sales dialogues[24].

Solimando et al.[25] and Sulzmann et al.[26] have proposed designs for BDI in Haskell. Both give suggestions to the definitions of agents, but don't give a complete and usable implementation.

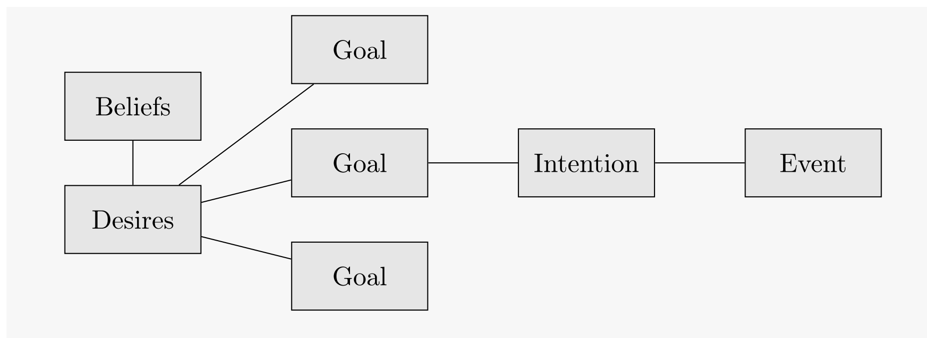

Figure 3.9: A map of connected terms, from desires to events.

The BDI framework is based on agents that act according to their beliefs and desires. An agent's actions are called events in the framework. Figure 3.9 shows the process to compute an event for an agent.. Using the beliefs, we can draft goals from desires. One of the goals becomes the current intention. The intention leads to an action for the agent. Following sections will discuss these concepts in more detail, as we define a working BDI framework to compare it with other frameworks.

3.5.1 Events

Events are all the possible actions an agent can possibly execute. In a BDI framework, the interactions of the client and pharmacist in a conversation translate to events for our agents. An *Event*, as defined in 3.10, is a pair of an interaction and a function that modifies our agents. When an agent acts out an event, the sentence of the interaction is printed, and the modifier is applied to the agents in the conversation. In theory, an agent should have sensors to detect events and interpret those events to update itself. One could argue an agent should manipulate itself according event, but in our case of conversing with two agents we define the events as interpreted by the receiving agent. This definition gives all freedom to the *modifier* to modify agents.

3.5.2 Beliefs

An agent has a collection of beliefs, which are facts and rules. A fact can be seen as a variable which has the value true or false, and a rule can be a logical implication. These facts and rules define what the agent thinks is true. Furthermore, the collection of beliefs can change during the conversation.

```
1 data Event = Event {
2   interaction :: Interaction,
3   modifier :: Agent → Agent
4 }
5   -- instances of Show and Eq omitted
```

Figure 3.10: Definition of *Event* data type definition.

```
6 data Belief = Fact String
7    | Or Belief Belief
8    | And Belief Belief
9    | ¬Belief
10   | If Belief Belief
11 deriving (Show, Eq)
12 memberBeliefs :: [Belief] → Belief → Bool
13 memberBeliefs bs b = or $ Data.List.map (flip memberBelief b) (applyIfs bs)
14 applyIfs :: [Belief] → [Belief]
15   -- belief checking
16 memberBelief :: Belief → Belief → Bool
17 memberBelief (Or x y)   b          = memberBelief x b ∨ memberBelief y b
18 memberBelief (And x y)  b          = memberBelief x b ∧ memberBelief y b
19 memberBelief (¬a′)      b          = ¬ (memberBelief a′ b)
20 memberBelief (Fact a)   (Fact b)   = a ≡ b
21 memberBelief (If a1 b1) (If a2 b2) = memberBelief a1 a2 ∧ memberBelief a2 b2
22 memberBelief a          b          = a ≡ b
```

Figure 3.11: *Belief* defined as a logical language.

Our definition of beliefs, in Figure 3.11, is minimal and could be extended to have temporal operators like: necessarily and possibly. We can define facts and rules using the *Fact* and *If* constructors respectively. We can negate beliefs with $\neg\cdot$ and combine two beliefs using the *Or* and *And* constructors. Secondly, we define membership of a certain belief in a collection of beliefs. Every agent maintains a collection of its beliefs. Membership combines facts and implication rules to deduce new beliefs, takes negation in account and respects the *And* and *Or* operators, while checking an belief to be part of an agents beliefs.

3.5.3 Desires

A desires defines the connection between a belief and an event. An action should only be selected for execution when the agent beliefs that doing so is is useful and feasible. When an agent believes that a desire will have a positive outcome, the desire will be selected as a goal.

Figure 3.12 shows the definition of the *Desire* data type and an example desire. A *Desire* is the maps a *Belief* to an *Event*. The belief value of a *Desire* is the requirement for the desire to be selected as a goal. In the example we define when the event for the *I.iassume* interaction becomes a goal, by defining

```
23 data Desire = Desire Belief Event
24   deriving (Show, Eq)
25 exDesire = Desire (And (And (Fact "client-wants-drug")
26                             (Fact "pharmacist-give-drug"))
27                           (¬(Fact "friendly-contact")))
28                      (Event I.iassume (addFact "client-has-drug"))
```

Figure 3.12: *Desire* data type definition and an example desire.

the required belief.

Every time an agent is asked (or allowed) to interact, it will use the agent's desires to select goals. We can define a function that selects goals from desires. A function *allGoals* would have the type $[Desire] \rightarrow [Belief] \rightarrow [Desire]$. The function returns a list of applicable goals, but we want a single *Event* to be executed.

3.5.4 Intentions

An agent's intention is the event which is believed to have the best result. Because the selection of goals often will not lead to a single event, we need to define how to select the intention of an agent from a list of goals. When relating to the agent that wanted to return home, it might have two reasons to go home. It might believe that it can go home for recharging and think that at home it would be safer. To see how which goal is most important for an agent to execute, we build up a weighted list. The weight is equal to the number of memberships of a belief in the agent's beliefs. In Figure 3.13 we define how the weighted list is created. If the *Belief* of a desire is member of the agent's beliefs we add the goal to the list, or if it is already in the list we increase the weight by one.

```
29 allGoals' :: [Desire] → [Belief] → [(Int, Event)] → [(Int, Event)]
30 allGoals' [] _ acc = acc
31 allGoals' ((Desire b e) : ds) bss acc = allGoals' ds bss (
32                                    if memberBeliefs bss b
33                                    then (addNumList e acc)
34                                    else acc)
35 where addNumList :: Eq a ⇒ a → [(Int, a)] → [(Int, a)]
36   addNumList x []                          = [(0, x)]
37   addNumList x (z@(i, y) : ys) | x ≡ y      = (i + 1, y) : ys
38                                | otherwise = z : (addNumList x ys)
```

Figure 3.13: Function for building a weighted list, which encodes how strongly a desire is believed to be a goal.

By sorting the returned list of *allGoals′* in descending order on the value of its weight, the first equally weighted goals are most important. We remove all the goals that have lower weight than the first item in the sorted list. If there is one goal left, it is the selected intention. For example, if we have three goals

in front of the sorted list of the same weight, we could accept all of these as intentions. If we accepted multiple goals in the resulting list, we can take an arbitrary goal as intention. The agent can execute the event of the intention that is selected.

3.5.5 Agents

Figure 3.14 shows a definition of agents. We define the agent's beliefs as a list of *Belief* items. The agent's desires is a list of *Desire* values.

```
39 data Agent = Agent {
40     beliefs :: [Belief],
41     desires :: [Desire]
42 } deriving (Show, Eq)
```

Figure 3.14: The BDI definition of an agent.

Specifying agents in a BDI framework requires an investment of effort because besides defining interactions, it involves defining beliefs and desires for every agent. The beliefs should enable desires to become goals, and the events of the desires need to adapt beliefs and/or desires such that new interactions come in to play. Effort is needed to have fine control over the conversation, because the flow of changing beliefs should be restricted such that desires only become goals when required. To keep the conversation going, we have to define the desires such that the continuity is guaranteed. In the example code below, in figure 3.15, we can see how lengthy it is to encode the first choice of the pharmacist. The first four options are easily described with just *And*, $\neg\cdot$ and *Fact* constructors. The helper functions *addFact* and *remFact* respectively add and remove values from the agents' beliefs. For brevity, I have excluded the second choice for the pharmacist as well as the whole definition for the client.

We describe a single desire for every interaction in the conversation. The *Belief* data type allows us to define two beliefs which lead to the same event, using the *Or* operator. Furthermore, complex beliefs can be build with the *And*, which is used extensively in the example. Although the definition for a couple of interactions might seem extensive, we don't define more than the desires and beliefs. When defining a large conversation, BDI might show efficient and expressive.

Let's apply a change in the options for the pharmacist. We want to give the pharmacist the option to ask the client to focus on the conversation by saying: `"I have the impression that you are absent-minded?"`. Figure 3.16 shows how we add a desire. Adding the desire to the pharmacist is easy, we take the previous desires and use the constructor of lists to add another desire. Next, we want to make sure that the desire is selected to be a goal, by setting the `"client-absent"` fact. For this we define a function that finds a desire with the right interaction, and adds the setting of a fact to the event of that desire. Dynamically changing agents is possible, but the writer has to make sure a new desire will become a goal. Moreover, every modification would require an change for the reacting agent. After adding the new interaction, the client needs to react to `"ask-absent"`.

```
43 pharmacistDesires =
44   [(Desire (And (Fact "pharmacist-has-drug")
45                (¬(Fact "client-has-drug")))
46              (Event I.yourmetformin
47                     (addFact "pharmacist-give-drug")))
48   ,(Desire (And (Fact "pharmacist-has-drug")
49                (¬(Fact "client-has-drug")))
50              (Event I.yourdrugs
51                     (  addFact "pharmacist-give-drug"
52                      ∘ addFact "friendly-contact")))
53   ,(Desire (And (Fact "ask-medical-information")
54                (And (¬(Fact "client-has-drug"))
55                     (¬(Fact "ask-generalpracticioner"))))
56              (Event I.practitionertellyou
57                     (  addFact "ask-generalpracticioner"
58                      ∘ remFact "ask-medical-information")))
59   ,(Desire (And (Fact "ask-medical-information")
60                (And (¬(Fact "client-has-drug"))
61                     (¬(Fact "ask-firsttime"))))
62              (Event I.correctfirsttime
63                     (  addFact "ask-firsttime"
64                      ∘ remFact "ask-medical-information")))]
65 pharmacistBeliefs = [(Fact "pharmacist-has-drug")
66                     ,(Fact "ask-medical-information")]
67 pharmacist = Agent pharmacistBeliefs pharmacistDesires
```

Figure 3.15: Defining the example conversation for one of the agents in BDI framework.

```
68 pharmacistDesires′ = newDesire : pharmacistDesires
69   where newDesire = Desire (Fact "client-absent")
70                            (Event I.absentminded
71                                   (addFact "ask-absent"))
72 addFactDesire :: I.Interaction → String → [Desire] → [Desire]
73 addFactDesire _ _ []     = []
74 addFactDesire i f (d@(Desire b e) : ds)
75   | (interaction e) ≡ i = d′ : ds
76   | otherwise           = d : (addFactDesire i f ds)
77   where d′ = Desire b (Event i ((modifier e) ∘ addFact f))
78 pharmacistDesires″ = addFactDesire I.correctfirsttime
79                                    "client-absent"
80                                    pharmacistDesires′
81 pharmacist′ = Agent pharmacistBeliefs pharmacistDesires″
```

Figure 3.16: Adding an option to the pharmacist’s desires.

3.5.6 Conversation

To allow two agents to communicate, we use the Conversation data type as defined in Figure 3.17, to distinguish the two actors. Furthermore we keep track of the interactions executed in the conversation, in a list of executed events. As a convention, we assume the first agent is active, and the second reactive. When we want the conversation to proceed, we can take the intentions of the active agent. If we swap the two agents after an event has happened, we can recursively use this function to simulate a dialogue until an agent has no goals to achieve or events to intend. Using the *pharmacist* definition given earlier and the omitted defintion of the client, we define a conversation. In our conversation the pharmacist starts the conversation. The list of events of this conversation is empty, which translates to the situation where the agents do not know anything about each other.

```
82 data Conversation = Conversation {
83    active   :: Agent,
84    reactive :: Agent,
85    events   :: [Event]
86 } deriving Show
87 swapAgent :: Conversation → Conversation
88 swapAgent c = c {active = reactive c
89                 , reactive = active c}
90 conversation = Conversation pharmacist client []
```

Figure 3.17: Definition of *Conversation* for a BDI framework between a player and a NPC

3.5.7 Overview

BDI allows the editor to enrich interactions with the effect that they have on the agents. This possibility comes with the cost to define sometimes elaborate logical statements as beliefs and modifier functions in events. For a single choice, and a follow–up reaction we have defined more than ten different beliefs that are then coupled with their respective events to create goals. The higher structure between interactions is defined loosely in desires, which makes them less readable compared to dialog trees. It requires more effort to define beliefs, desires and intentions for a whole scenario than in a dialog tree, but a framework based on a BDI model is better suited for the reuse of interactions.

In some ways the framework is not optimal for our specific problem. We need to define agents for the client and pharmacist, even though the pharmacist is not simulated but is controlled by the player. We have no need for the selection of an intention of the pharmacist, as we want the player to pick an intention from the goals. Moreover, the framework does not restrict the agent for the NPC to have at most one intention. The client agent needs to come up with a single intention, while the pharmacist should return four goals to allow the player to choose from. We have some restrictions to our problem that could make our problem easier to deal with, but every time an agent is asked to interact, functions are used

to select goals and weigh them. We define desires for an agent, most of which are only applicable in one or a few places of the conversation. Although, that a desire is applicable in a specific phase, the desire will be checked for goal selection throughout the conversation.

Modifying agents of a BDI model is quite challenging, every change in interactions has to be implemented in the beliefs and desires. It is the editor's task to make sure that the weight of that goal is the highest at some point in the conversation. All these changes involved in a single change of the scenario make it error prone for an editor to redefine conversation in a BDI framework. Maintaining and modifying existing agents is hard since all the information is scattered over the agent's beliefs and desires.

An agent possibly has multiple goals that weigh equally, in that case an intention is chosen arbitrarily by the system. Because of this non-deterministic uncertainty, we cannot predict how the feedback is built up. The problem with giving feedback on a non–deterministic path of a conversation is that fine control is lost. Since we cannot predict the outcome of a run of a conversation, we cannot be sure what feedback is actually built up, or is not built up. If we want to give good feedback in every possible outcome, we could give feedback on the beliefs of the agent at the end of the exercise. Other ways to give localized feedback, feedback that discusses a specific part of the conversation, could be to look at the beliefs. Beliefs are often (re)used and changed extensively throughout the conversation, which makes giving localized feedback difficult. We can connect values on outcome measures to events and analyse what the scoring of the player is, relative to the best and worse case scenario. Using list of events that have happened, we can compute feedback, but for a comparison with the best and worst case scenario we would have to compute all possible events. It is costly to compute all possible alternative events, since it involves calculating the goals at every agent's event of the conversation. Overall the framework is effective to be used in a serious game.

3.6 Domain reasoners

The domain reasoner framework is based on the mathematical view on problems. A framework for domain reasoners specifies a problem, and rules that rewrite the problem towards a solution[27]. The problem is described using a term. The mathematical term is often an equation, for example. In communication we should describe the current state of the conversation. In our scenario the client and pharmacist try to reach their goals, while talking. The interactions between client and pharmacist are the transformations on the conversation state.

Using a domain reasoner we develop interactions that affect the conversation. Interactions in the domain of communications become rewrite rules[28], described in a domain specific language.[29] Every rule has an effect on the conversation state, thus a rule solves a part of the problem. The utterances of the player and NPC are rules and rewrite the conversation state. The higher structure of transformation rules is described in a strategy. Strategies can solve whole problems, or in our communication domain can complete a whole conversation. A visualization of the strategy structure might look like that of a dialog tree, except that there is no limitation to how many edges a node must have. Furthermore, the transformations on a state are similar to modifiers of agents in BDI.

First we define the state in Section 3.6.1. In Section 3.6.2 we define ruled, which rewrite the state. Section 3.6.3 defines a strategy which uses the rewrite rules. An overview and conclusion is given in Section 3.6.4

3.6.1 State

A domain reasoner applies rewrite rules to the matter of a problem. In mathematics for instance, the problem matter is an equation. We need to define the problem matter of a conversation, which we name as a state.

The knowledge of what is asked and answered is built up in our state. While the player and NPC are interacting, knowledge is shared via utterances. A communication state should capture the knowledge that is built up during a conversation. Furthermore, we define parameters that describe the state of a client, and the learners scoring on outcome measures. The state can contain all information that is built up in a conversation. For example, we could define how the player has affected the relation with the NPC in specific client parameters. It is very possible that it is desirable to build up feedback information that can be used at any time of an exercise to allow the learner to reflect on its choices.

```
1 data State = S
2   {knowledge :: Map String StateValue
3     -- , clientParameters :: SomeDatatype
4     -- , learnersGoals :: SomeDatatype
5   } deriving Show
6 data StateValue = Inactive
7                 | Active
8                 | Completed
9   deriving (Show, Eq)
```

Figure 3.18: domain reasoners definition of state.

In Figure 3.18, we define a mapping from *String*s to *StateValue* elements. The state value elements contain pieces of knowledge from interactions. For simplicity, we comment the suggested parameters that could give more detailed feedback. The parameters allow for the comparison of rules on how they affect the client–pharmacist relation and how a learner scores when using a rule. This definition allows us to define the example scenario without feedback. The definition for *StateValue* allows us to define three states of knowledge. An mapping to an *Inactive* value means that a subject is not yet discussed. If the value is *Active*, a question concerning the subject is asked. And lastly, *Completed* knowledge mappings are recognized as a sub–problem that is solved. If there is no mapping for a specific string, we say that it is the same as *Inactive*.

Figure 3.19 shows an state for an conversation. In the example the client has been introduced to the pharmacist, but he did not receive the medicine yet. Currently, the pharmacist is giving information on how to use the medcine. This is just one definition of the state, a domain reasoner does not restrict us to use a particular definition.

```
10 currentState = fromList [("told-name", Completed)
11                         ,("received-medicine", Inactive)
12                         ,("received-usage-info", Active)]
```

Figure 3.19: Some example state values.

3.6.2 Rules

Rules are used to modify the state values such that they reflect the conversation. Figure 3.20 shows the *Rule* data type definition, and a helpful wrapper to lift the rule constructor to the strategy data type. Rules are interactions with a guarding and modifying function. A guarding function checks if a rule is applicable in some version of the domain state. It is used to reduce the collection of all interactions to those that are valid options for the pharmacist in the current situation of the conversation. The modifier encodes the effect of the interaction on the state.

```
13 data Rule = R {
14   interaction :: Interaction,
15   guard :: State → Bool,
16   modifier :: State → State
17 }
```

Figure 3.20: Definition of a rule that wraps an interaction.

We can define *Rule*s as in Figure 3.21. A modifier updates knowledge in the state. These functions have the type *State* → *State*. A helper function such as *setActiveKnowledge* activates a state value in the knowledge map. We can chain functions that update the state, using the composition function (∘) from the Haskell prelude. Similarly we can define guards using helper functions. We encode all the knowledge needed to find a single rule in every possible state. Given an extended state definition for feedback purposes, the rules require more extended modifiers.

3.6.3 Strategy

In Figure 3.22, we define a small domain–specific, combinator language to describe the strategy. It is a subset of the language defined by Heeren et al.[30][31] To create a strategy for the example conversation, we should be able to sequence strategies, and allow choice between two strategies. We use them to implement one interaction from the pharmacist followed by a reaction from the patient. We can lift rules into the strategy language, by wrapping them in the *Atomic* constructor.

Using the earlier defined rules, we can define some strategy of sequences or choices of rules. We could sequence every path together and put choices where the pharmacist is given a choice between interactions. This would result in a tree–like structure, which is similar to a dialog tree. We improve on the tree–like

```
18 rYourmetformin = R I.yourmetformin
19                      (λ_. True)
20                      (setActiveKnowledge "Give-Metformin")
21 rYourdrugs     = R I.yourdrugs
22                      (λ_. True)
23                      (setActiveKnowledge "Give-Drug")
24 rIassume       = R   I.iassume
25                        (isActiveKnowledge "Give-Metformin")
26                          ((completeKnowledge "Give-Metformin")
27                           ∘ (setActiveKnowledge "Assume"))
28 rYesthanks     = R   I.yesthanks
29                        (isActiveKnowledge "Give-Drug")
30                          ((completeKnowledge "Give-Drug")
31                           ∘ (setActiveKnowledge "Thanks"))
```

Figure 3.21: The first interactions of the example scenario.

```
32 data Strategy = Strategy :⋆: Strategy
33               | Strategy :|: Strategy
34               | Fail
35               | Succeed
36               | Atomic Rule
37   deriving Show
```

Figure 3.22: Basic strategy data type definition for combining rules.

structure by using the fact that an NPC has no choice to make. Figure 3.23 shows how we can define the scenario. We pair an interaction of the pharmacist with an interaction of the client. Since between those two interactions, no choice can be made, we can fuse their guards and modifiers as if it is one rule. With *foldChoice* we can fold rules such that they are combined as a choice.

```
38 foldChoice :: [Strategy] → Strategy
39 foldChoice = Prelude.foldr ( :|: ) Fail
40 strategy = foldChoice [(sCorrectfirsttime      :⋆: sYes)
41                       ,(sYourmetformin         :⋆: sIassume)
42                       ,(sYourdrugs             :⋆: sYesthanks)
43                       ,(sPractitionertellyou   :⋆: sUsemedication)]
```

Figure 3.23: Defining a scenario strategy for the example.

Heeren et al.[32] have discussed the need and the means to adapt a domain reasoner in a mathematical setting. Fortunately, these means transfer to the communication domain. To add an extra part of a dialogue, a strategy is easily extended. To change an existing strategy, the ease of changing a scenario would depend on if it is a change to the state or just the rules. While changing one rule

would have little impact on the rest of the strategy, changing many can be very involved. Changing the state requires that the writer of the conversation reviews all existing rules. Adding a new choice option to a conversation shouldn't be hard since a new rule does not affect the existing rules.

For example, we add a new interaction pair extending a choice option. We can define two new rules, pair them and add them to the strategy without conflicting with the original strategy. In Figure 3.24, we add a new pair to an existing strategy. Since every rule is guarded and modifies the state in its own way, we can easily add or remove options without interfering with other parts of a conversation.

```
44 sSomeInteraction = Atomic $ R someInteraction (guardFunction)  (modifiers)
45 sSomeReaction    = Atomic $ R someReaction    (guardFunction') (modifiers')
46 strategy' = foldChoice (sSomeInteraction :⋆: sSomeReaction) : strategy
```

Figure 3.24: Adapting a domain reasoner: adding part of conversation to the strategy.

3.6.4 Overview

The amount of control is up to the writer of the exercise, because the definition of the state and rules don't limit the writer. The more control you want, the more detailed the state will have to be and the rules that will modify the state accordingly. Like in the framework based on a BDI model, the more effort invested in developing the exercise, the more control the writer will have.

Defining feedback for a scenario in a domain reasoner is no problem. We can attach values of outcome measures, like we would in a BDI framework. Furthermore, we can take a strategy and use the, at some moment, applicable rules to give feedback on that specific option. We could add custom feedback on certain rules in the conversation for localized and detailed feedback. The editor is given freedom to define the feedback that is desired.

A domain reasoner framework needs some investment in the definition of the correct state variables, but once the state is defined the framework brings efficiency and adaptability for rules. We define high and low level structure with strategies and guards, respectively. Because of the well–structured exercises, a domain reasoner's feedback generation is best of the three frameworks. A domain reasoner framework is readable, maintainable and changeable due to its concise definition. Concluding, a framework based on a domain reasonermeets the technical requirements.

3.7 Comparison

We have defined the three ways of describing conversation such that we can compare them on the following aspects. In an educational conversation simulation, we need to control which options are suggested to the learner and how the NPC reacts. If we can control which options are suggested when, we define that as fine control. We look for fine control over the conversation instead of

coarse because we want to define precise conversational exercises. The finest control possible is where the writer can manipulate every choice at any time in the conversation. Coarse control means that the writer directs the flow of conversation by indirection, losing direct control. A framework that requires many definitions to implement a scenario is less desirable than a framework that allows concise definitions. If changing existing exercises, scenarios or interactions require little work, it scores positively on the maintainability of the framework. A highly maintainable framework has separate definitions of interactions, and their structure. Lastly, we compare the frameworks on how we could compute feedback. All frameworks require enriching interactions with extra information to give feedback, but the way we are able to compute feedback is different.

Framework	Control	Usability	Maint. & Adapt.	Feedback
Dialog tree	+	+	−	±
BDI	±	−	±	±
Domain reasoner	+	±	+	+

Table 3.1: Comparison of three frameworks on control, usability, maintainability & adaptability and feedback generation.

Table 3.1 shows how the frameworks scored on the requirements. The symbols show how frameworks compare to each other. A ± symbol indicates a framework scored average, − and + indicate a lower and higher score, respectively.

3.7.1 Control

A dialog tree allows you to build a tree of node and edges, giving the editor full control. The control of domain reasoners and BDI models require more elaborate definitions to control the conversation. Both frameworks have a state data type that is modified by functions that are combined with an interaction. The reason that the domain reasoner scores higher is that the state of a domain reasoner is a single entity, whereas the BDI defines a state per agent. Furthermore, a domain reasoner finds applicable rules from the current strategy, based on the state. The BDI model defines a state per agent, and requires us to define beliefs and desires. From the beliefs and desires follows a list of goals, and one goal is computed to be executed.

We defined our dialog tree such that the player always has a list of choices, and the NPC precisely one. In a BDI model we have to put effort in ensuring that the NPC always has precisely one goal. Domain reasoners puts no restrictions on the amount of rules that are applicable, but we can define a strategy such that the NPC's reaction is coupled with the player's.

3.7.2 Usability

Although all frameworks are not too elaborate to read, understand and use, we consider a dialog tree to be most user–friendly. Interactions can be used without extra effort in edges and nodes and the whole conversation is easily read from the definitions. As said earlier, we invest extra effort per interaction by defining beliefs or guards, and a modifier function for BDI models and domain reasoners, respectively. The concepts that form a sequence of interactions require extra

information. In BDI models, we define a single collection of desires per agent. We use a logical language to describe the sequence of interactions. Domain reasoners allow us to define rules that are applicable to the state. The editor is not restricted to a logical language when defining the state and guards. Moreover, we can use strategies to define higher structures in the collection of rules. BDI is less usable due to its use of logic, and lack of higher structure when compared to the domain reasoner.

3.7.3 Maintainability and Adaptability

When adding new or updating erroneous interactions, we find that the dialog tree and the BDI models are not easily adapted. Changing a single interaction in a dialog tree is straight forward, but making changes in the sequence of interactions involve more effort. As showed earlier, changing a desire in a BDI model also involves the desire that lead to and from that desire to be updated. When having a conversation with two agents, this involves the updating of desires of both agents. The continuity of the conversation lies in desires, being selected as goals, and allowing following desires to be selected. It is easier for an editor to change a domain reasoner, because a strategy can be manipulated easily and new (sets of) rules can be added without conflicting with others. Furthermore, changing a guard or modifier only affects a single rule, thus allowing changes to be made without concern to rest of the strategy.

3.7.4 Generating Feedback

Giving feedback using a dialog tree is difficult, since the framework's scenario description needs to be enriched with custom feedback on every path. We can build up feedback for every option of the pharmacist. Frameworks based on BDI models and domain reasoners allow the generation of feedback based on the values of the agent and state, respectively. Comparing interactions on these parameters, can be used to show localized feedback to the player. The paths in a conversation are not set in a BDI model, we cannot give detailed feedback based on a single path in the conversation. A domain reasoner combines the possibilities of dialog trees and BDI models. It allows the editor to add feedback parameters in the definition of the state, and compare interactions similar to a BDI model. Furthermore, we can define feedback on (sub–) strategies, similar to dialog trees. Adding extra information about the conversation is as simple as enlarging the state, and allowing rules to modify that state accordingly. A domain reasoner offers the most possibilities to generate feedback.

3.7.5 Concise Implementation

The amount of work that the editor would have to invest to describe a conversation differs greatly. All frameworks require the editor to write the actions of the players. When using a dialog tree, the editor has to map the actions to nodes and edges to create the conversation. In BDI, the editor defines the beliefs and desires, and defines how actions affect beliefs and desires of an agent. Furthermore, effort is required for restricting the NPC in one goal, and making sure the player can be presented an set of possible intentions. In a domain reasoner, the editor defines the state parameters, and how actions modify these

parameters. Moreover, the editor enrich the actions with guarding functions that define in what state an action is applicable. These enriched actions are then combined into a strategy.

Framework	Data.	Const.	Implementing a scenario		
			Functions	Const.	Combined
Dialog tree	3	4	0	17	17
BDI	5	9	7	125	132
Domain reasoner	5	9	55	1 *	56

Table 3.2: Counts of data types and their constructors, and the number of helper functions and constructors used to define the example scenario.

Table 3.2 shows the counts of data types, constructors, helper functions and constructors used for the three frameworks. For the size of the table, the headers are shortened. The first row is the count of data types that define a framework, with the header 'Data.'. Next under the name 'Const.', we count the number of constructors in the data types from the first row. Third from the left is the number of functions that were used in defining the example scenario. Forth states the number of constructors to define the scenario. Lastly, for ease of comparing, the summed value of the third and forth values.

When we compare the number of data types and constructors needed to define the framework, a dialog tree is less complex than the other two. BDI models and domain reasoners are equal in numbers which means they are similar in the complexity of their data structures. The number of functions and constructors used to define the example scenario differ greatly between the frameworks. The lesser complex framework is again the dialog tree, the definition of edges and nodes requires no helper functions and only seventeen constructors. The framework using a BDI model is most complex, because it uses many constructors to define the desires. A framework that delivers domain reasoners uses helper functions extensively, but is less complex overall when compared to the BDI framework.

Moreover, we can compare the complexity of the frameworks by looking at the functions that allow us to use the scenario description in an application. These are different from the helper functions to define scenario, since they allow the scenario to be used in a client application, for example. For using a dialog tree, we could make do with a common traversal function that follows the edges to the leafs of the tree. Again the dialog tree is less complex compared to the other two. The BDI model uses functions to find goals from the beliefs and desires. On top of that, we need to facilitate the modification of an agent beliefs and desires. Similar to functions in BDI, a domain reasoner requires functions to guard a rule and modify the state. A domain reasoner requires functions to be defined that handle guard– and modify–functions correctly. Concluding, a framework using a dialog tree is least verbose, followed by a domain reasoner and the BDI model is the most verbose.

3.8 Conclusion

Although the dialog tree allows a clear way of describing conversation, compared to the other frameworks it is lacking. The simple structure and fine control do not outweigh the high costs of effort for adapting and maintaining an exercise defined in a dialog tree. Furthermore, the framework captures no additional information that might assist the generation of feedback.

A framework that uses a BDI model allows an NPC to have an intelligent way of computing its own actions. In a BDI model, each agent is uniquely influenced by the current situation, which gives a sense of intelligence. In a game that is purely revolving around conversation, the actions of the NPC are limited to talking. In the setting of a player choosing from interactions, and an NPC that has only one option, defining two intelligent agents and then limiting them seems superfluous. Defining a detailed intelligent agent for the player and NPC seems a lot of work for little profit. Feedback is an important factor of learning, using a BDI model, extra effort is needed to give localized feedback.

A framework for domain reasoners is similar to a that of a BDI model to the extent of enriching interactions with function that modify some state. The BDI model requires the editor to let two agents manipulate each other during the conversation, while a domain reasoner manages with one state. Because of the single definition of a state data type, it is easier to define modify functions for it when compared with BDI's desires. Besides a state and rules, a domain reasoner incorporates a high structure definition in strategies, similar to a dialog tree. Lastly, the domain reasoner has the combined options of a BDI model and a dialog tree to compute feedback. Furthermore, we can use the single state to keep track of feedback, rather than having to analyse two agents, in BDI. A domain reasoner is the best option for defining exercises in a communication game between a player and a NPC.

Chapter 4

Designing Artificial Intelligence for 'Communicate!'

Comparing alternative frameworks led to the choice to proceed with the IDEAS framework that was developed for serving exercises and generating feedback. We can define domain reasoners on top of the IDEAS framework. The framework offers services that allow a client application to use exercises. In section 4.1, we walk through the framework's current usages and its accompanying paradigm. We will look how we can translate the concepts of the earlier usages to the domain of communication. Section 4.2 documents the implementation of the communication domain and the choices during the implementation process.

4.1 IDEAS framework

IDEAS is a framework for developing domain reasoners that generate intelligent feedback[33]. Rewrite strategies describe how an exercise can be solved incrementally. The goal of the framework is providing detailed feedback. A progression of steps from a strategy applied to a problem allows diagnosis of the current state of a problem[30]. Diagnosis leads to the ability to give hints to the player, and give a comparison of alternative applicable rules. That results in feedback that is tailored to the learner's way of trying to solve an exercise.

4.1.1 Existing applications of IDEAS

IDEAS has been used in a couple of applications so far. Domain reasoners for many kinds of exercises have been developed. The IDEAS framework is used in a learning tool for rewriting logical expressions to disjunctive normal form. Furthermore, domain reasoners for solving exercises in linear, quadratic and higher-degree equations and in-equations have been developed. They allow simplifying and evaluating fractions, expressions using powers and square roots, etcetera. Reasoners have been developed for linear algebra, including Gaussian elimination and Gram–Schmidt, and solving systems of linear equations. After

developing reasoners for mathematical domains, a functional programming tutor named Ask-Elle[34][8] was developed. Ask-Elle allows a learner to program Haskell functions with the help of a domain reasoner.

4.1.2 Paradigm

The framework employs services for client–applications. Tools, tutors and game applications can make service–calls to a variety of functions to accompany the specific presentation of exercises. Using IDEAS does not limit developers and designers of a client–application. The framework analyses the current problem, it can produce the next possible steps or give a whole solution. Using the framework, creators of a game can define what kind of steps are offered to the player. Developers of a client–application can choose the way in which feedback is given to the player[30].

The IDEAS package[35] is released on the on-line package repository Hackage. Besides basic services, the list of services includes those handling feedback scripts. The basic services generally offer enough functionality to use an exercise. Feedback in textual form can be gained by the feedback script services.

Some of the most important services offered by the framework are these basic services[1]:

1. $\mathit{ready} :: \mathit{State}\ a \rightarrow \mathit{Bool}$
   Returns *True* if the current state of the problem is a solution.

2. $\mathit{allfirsts} :: \mathit{State}\ a \rightarrow \mathit{Either}\ \mathit{String}\ [(\mathit{StepInfo}\ a, \mathit{State}\ a)]$
   allfirst either returns all rules for the given State with their location, environment and the resulting state, or an error message when there are no first rules.

3. $\mathit{allapplications} :: \mathit{State}\ a \rightarrow [(\mathit{Rule}\ (\mathit{Context}\ a), \mathit{Location}, \mathit{State}\ a)]$
   Returns all currently applicable rules with their location, environment and the resulting state. Other than *allfirsts*, the function uses guards to filter out inapplicable rules.

4. $\mathit{apply} :: \mathit{Rule}\ (\mathit{Context}\ a) \rightarrow \mathit{Location} \rightarrow \mathit{Environment} \rightarrow \mathit{State}\ a \rightarrow \mathit{Either}\ \mathit{String}\ (\mathit{State}\ a)$
   Applies a rewrite rule to a state.

Using the basic services we can manipulate and apply rules to a state, but we need more for a game to work. When the game starts off a conversation, we need to select the exercise and request the possible options for a player. By using the function *exerciselist*, we request the list of exercises which allows to select an exercise. We can find possible options by using the functions *allapplications* and *allfirsts*. To recognize the ending of a level in a game, we can use the function *ready* to analyse if there are more possible steps. The services return identifiers which can be connected to the sentences of rules and feedback.

[1] `http://hackage.haskell.org/package/ideas-1.1/docs/Ideas-Service-BasicServices.html`

4.2 Implementation

The framework can be compiled to a web–service application, which can be deployed as an CGI executable. The domain reasoner is used in many web–services. In the next subsections we show the steps to define a domain reasoner based on the example pharmacy domain. Some definitions are omitted for brevity.

For a minimal definition we just need an exercise and the standard set of services. Figure 4.1 shows how we define a minimal reasoner for the example conversation in the pharmacy domain. The example exercise is imported as part of the pharmacy package, under the qualified name *Pharmacy*. The most interesting field of the *DomainReasoner* is *exercises*. In our case it is a list with a single element since we define one exercise.

```
1 ideasPharmacy :: DomainReasoner
2 ideasPharmacy = (newDomainReasoner "ideas.pharmacy")
3                   {exercises = [Some Pharmacy.exercise]
4                   , services  = metaServiceList ideasPharmacy ++ serviceList}
```

Figure 4.1: An example domain reasoner in the IDEAS framework.

The *services* are defined using standard services, using *metaServiceList* and *serviceList* to build up basic and reflective services.[2] The four other items: *views*, *aliases*, *scripts* and *testSuite* are currently unused. These are not essential for our purpose of showing the design of the domain reasoner.

An implementation of a domain reasoner using the framework requires the definition of exercises. Exercises are problems concerning some notion of a term which can be modified until it is solved. An example is an equation for a mathematical domain. Furthermore we define rules that modify the state and a strategy which is a combination of rules. The following sections explain the implementation of the communication domain geared for educational exercises.

4.2.1 Communication Exercises

The *exercises* field in the *DomainReasoner* record contains exercises like the one defined in Figure 4.2. The *Exercise* data type holds all necessary information for exercises. We instantiate the polymorphic type *a* of *Exercise* with a *State*, which will be explained later.

The *exerciseId* and *status* give information to find a specific exercise. The identifier uniquely identifies every exercise such that it can be used in a service call. The status is an indication of how far developed the exercise is. The *parser* and *prettyPrinter* are needed to create a value of type *a* from a *String*, and a *String* from a value of type *a*, respectively. Custom parsers and printers are sometimes necessary, but for our rather straight forward domain we can use the Haskell built–in derived classes *Show* and *Read*. The *strategy* value defines how to solve the exercises and which rules it can use. Lastly, as an example exercise,

[2] `http://hackage.haskell.org/package/ideas-1.1/docs/Ideas-Service-ServiceList.html`

```
 5 exercise :: Exercise Conversation
 6 exercise = makeExercise
 7   { exerciseId    = describe "Casus 1; Example exercise for pharmacy" $
 8                       newId "pharmacy.casus1"
 9   , status        = Stable
10   , parser        = Right ∘ read
11   , prettyPrinter = show
12   , strategy      = liftToContext strategy1
13   , examples      = [(VeryEasy, emptyConversation)]
14   }
```

Figure 4.2: Example definition of a single exercise.

we give the *emptyState*, which holds no specific state information. We leave out most of the values of the *Exercise* record because we don't need them or we can use the default implementations of *makeExercise*.

4.2.2 Conversation

In Figure 4.3 we define the three elements of the state. For conversation exercises, the state is the *a*–term of the *Exercise a*. The state relates to an expression in the mathematical domain, for example an equation that can be rewritten. A term in the framework is the data that is rewritten by every step, and a rule modifies the term.

```
15   -- The parameters of a client, which reflect the current state of the client.
16 data ClientParameters a = CP
17   { cpSatisfaction  :: a
18   , cpTrust         :: a
19   , cpInformation   :: a
20   , cpAnxiety       :: a
21   , cpAggression    :: a
22   } deriving (Show, Eq, Read)
23   -- The learning goals reflect the score of the student.
24 data LearningGoals a     = LG
25   { lgContact        :: a
26   , lgProblem        :: a
27   , lgDecision       :: a
28   , lgSubjectmatter  :: a
29   , lgClear          :: a
30   , lgStructure      :: a
31   } deriving (Show, Eq, Read)
32 type ConversationKnowledge a = M.Map String a
```

Figure 4.3: Definitions of the (data) types for the client parameters, learning goals and conversation knowledge.

The client parameters hold a predefined set of values to describe the client.

Based on requirements of the specific conversations in the pharmacy, we have chosen the values to be satisfaction, trust, information, anxiety and aggression. As rules are applied to the state, they might increase or decrease the values of the client parameters. Feedback on the relation between the client and pharmacist can be generated using these parameters. Using a record notation for the *ClientParameters a* allows modification of the parameters.

Like the *ClientParameters*, the *LearningGoals* data type is defined using ecord notation. These values describe the effectiveness of the learner. While two rules might achieve the same effect on the client, one might be more focused on the problem and therapy and the other more on strict structure of a conversation. One is not better than the other per sé, but when looking at the whole conversation, this might be the perfect place to follow the structure. Neglecting one of the goals will result in a lower overall score.

The last of the three elements is the *ConversationKnowledge*. It is 'abstractly' defined as a *Map* from string keys to integer values. It holds the information of a conversation. For instance, we might start off with a client with the desire to ask a question, and until that question is asked a client is not happy with leaving. The most common use of the conversation knowledge is to describe which questions are asked, so that the player and NPC can react to each other. Although a mapping from *String* to a value of type *a* does not provide type–based pre–compilation checks on the keys, it is a usable and readable solution.

```
33   -- State values
34 data GenState a = State
35   {cKnow  :: ConversationKnowledge a
36   , cParam :: ClientParameters a
37   , lGoals  :: LearningGoals a
38   } deriving (Show, Eq, Read)
```

Figure 4.4: Generalized *State* definition

Combining the *ClientParameters*, *LearningGoals* and *ConversationKnowledge* leads to the definition of a generalized state in figure 4.4. The *GenState a* data type can be reused for similar communication domains.

```
39 instance IsTerm a ⇒ IsTerm (GenState a)
40 toTerm :: IsTerm a ⇒                    a → Term
41 fromTerm :: IsTerm a, MonadPlus m ⇒ Term → m a
```

Figure 4.5: Type definitions of the functions *toTerm* and *fromTerm* as part of the *IsTerm* class.

Figure 4.5 shows the type definitions of the functions in the *IsTerm* class. *Term*s are used to describe a state definition of any domain, uniformly. Every value of *a* that is an instance of *IsTerm* turned into a *Term* and from *Term* to *a* value using the functions *toTerm* and *fromTerm* respectively. A uniform description of types is needed for the serialization of data types used in the requests and replies for the web service.

We define a *Conversation* as the term for our communication exercises. We instantiate the *GenState* with *Int* values. The definition in Figure 4.6 shows the type name definition, and a function that creates an empty *Conversation*.

```
42 type Conversation = GenState Int
43   -- identity state
44 emptyConversation :: Conversation
45 emptyConversation = State (M.empty)
46                           (idClientParameters)
47                           (idLearningGoals)
```

Figure 4.6: The *Conversation* data type as the term of the communications domain for pharmacy.

The *Conversation* is the subject of our guarding and modifying functions. The writer of a scenario is aided by readable codes. We define helper–functions for guarding and modifying *Conversations*. The aim is that the editor does not need to worry about the implementation, but can focus on the semantics of the rewrite rules. Figure 4.7 shows a modifing function and guard function that work together. The function *happens* modifies the *Conversation* such that the function *happened* can guard a rewrite rule on the same string.

```
48   -- Sets everything to 0, except for the new value.
49 happens :: String → Conversation → Conversation
50 happens σ c = c {cKnow = (M.insert σ 1) (cKnow c)}
51   -- Check if some thing has happened.
52 happened :: String → Conversation → Bool
53 happened σ c = checkValue σ 1 c
```

Figure 4.7: Helper–functions for guarding and modifying the Conversational Knowledge.

The above functions address the *ConversationKnowledge* of a *Conversation*, but more importantly we want to build up feedback parameters. Figure 4.8 shows a modifying function that adds an integer to the client parameter satisfaction. For every client parameter and learning goal we have defined these functions. We can add or subtract a value by using positive or negative numbers, respectively.

```
54   -- Modifier function
55 satisfaction :: Int → Conversation → Conversation
56 satisfaction i gs =
57   gs {cParam = (cParam gs)
58     {cpSatisfaction = (cpSatisfaction (cParam gs)) + i}}
```

Figure 4.8: A helper–function that can modify the client's satisfaction parameter.

Note that the guard functions can be composed by Haskell functions like ($\vee$) and ($\wedge$). Furthermore, we can compose modifier functions using Haskell's ($\circ$). The function *happens* should only be used once in a modifying function. Composing the *happens* function with another *happens* function will result in the same, as just having the first. Using *happens* will set all values to 0 before adding another value, and set it to 1. When applying two after each other, only the latter persists in the mapping.

4.2.3 Sentences

Besides having readable and composable helper–functions that work with the *Conversation* data type, we define the functions to create sentences as *Rule*s for transforming a *Conversation*. We apply syntactic sugar to the rewrite rules, by defining a *Sentence* to be a *Rule* of *Conversation*. To create sentences, we define smart constructors that hide the technical details for the editor.

```
59 type Sentence = Rule Conversation
60 makeSentenceExt :: String
61                  → String
62                  → (Conversation → Bool)
63                  → (Conversation → Conversation)
64                  → Sentence
65 makeSentenceExt identifier sentence guard modifier = describe sentence
66    $ makeRule identifier
67             (λx. if    guard x
68                  then Just (modifier x)
69                  else Nothing)
70 makeSentence :: String
71              → (Conversation → Bool)
72              → (Conversation → Conversation)
73              → Sentence
74 makeSentence = makeSentenceExt ""
75 makeSentenceSucceed :: String → Sentence
76 makeSentenceSucceed σ = makeSentence σ true id
```

Figure 4.9: Type definition and constructor functions for *Sentence*s.

Figure 4.9 shows the function that creates sentences. The main function is *makeSentenceExt*, that allows to specify the identifier, along side with the actual sentence string, guard and modifier. Often the identifier is not specified by the editor of the scenario. The function *makeSentence* allows the editor to write sentences without worrying about the identifier. Later on in the strategy, we will have to make up for the ease of defining sentences, by traversing the strategy and making sure that all sentences have a unique identifier. The last definition, *makeSentenceSucceed* is the definition of an unguarded sentence that has no effect to the conversation. These can be used when a sentence is always applicable and has no effect on the *Conversation*.

The function uses *describe* and *makeRule* to define a rule with a sentence coupled to the rule. This way a rule and a sentence are not tied together, but

the description is a label to a rule. The rule is built as a transformation with the type $\mathit{MakeTrans}\ a \Rightarrow a \rightarrow f\ a$.

Figure 4.10 shows an example rule for a pharmacist. The first argument in the definition is the interaction's sentence in textual form. In this case we ask the client what she knows about a particular drug.

The second argument is the guarding function, which is required to be of type ($\mathit{Conversation} \rightarrow \mathit{Bool}$). In the example, we check if the event `"hand over drug"` has happened. Notice that the string values of *happens* and *happened* have been defined as string constants.

Lastly, we define a function that can manipulate the conversation knowledge, client parameters and learning goals. The modifier function is a composition of smaller functions. The first function modifies the conversation knowledge. The rest of the functions update the client parameters and learning goals. The client parameters and learning goals are modified such that they reflect the effect of the single sentence. Since all functions that modify the conversations have the type $\mathit{Conversation} \rightarrow \mathit{Conversation}$, we can compose them together using function composition.

```
77   -- string synonyms
78 keyHandOverDrug      = "hand over drug"
79 keyPhysiscistTellYou = "physicist tell you?"
80 personalAskKnowledge2 :: Sentence
81 personalAskKnowledge2 = makeSentence
82   (     "Miss Darcy, what did your general practitioner "
83      ++ "tell you about the metformin?")
84   (happened keyHandOverDrug)
85   (  (happens keyPhysiscistTellYou)
86    ∘ (satisfaction 1)
87    ∘ (trust 1)
88    ∘ (contact 1)
89    ∘ (problem 1)
90    ∘ (structure 1)
91   )
```

Figure 4.10: Definition of an example sentence; an interaction for a pharmacist.

A conversation consists of many interactions. Creating sentences using the helper functions and a constructor function like *makeSentence* decreases the amount of work to implement a conversation. Also the ease of creating modifiers and guarding functions helps in developing a conversation. Moreover, the readability of sentence declarations is increased.

We allow scenario editors to create concise sentences. We want to hide some of the technical details such as identifiers. By allowing the editor to create rules without identifiers, we postpone setting the identifier of a rule. We define a function that provides unidentifiable rules with an unique identifier. Figure 4.11 shows the definition of *labelRule*, which prepends a rule with a string and provides a unique identifier to rules if necessary. Using the monad *State*, we define a counter such that we can create identifiers. When a rule is not yet identifiable, i.e. as a product of *makeSentence*, we use the number from the

counter to create a unique identifier.

```
92  labelRule :: String → Rule a → State Int (Rule a)
93  labelRule [ ]  r | notEmptyId r = return r
94  labelRule pf  r | notEmptyId r = return $ appId pf  r
95  labelRule pf  r | otherwise        =
96    do i ← get
97         put (i + 1)
98         return $ appId pf $ appId (show i) r
99  appId :: String → Rule a → Rule a
100 appId i = changeId ((#) (newId i))
101 notEmptyId :: HasId a ⇒ a → Bool
102 notEmptyId r = ¬ $ isEmptyId (getId r)
```

Figure 4.11: Definition of *labelRule* to append a *Rule*'s identifier.

Note that chaining two *labelRules* functions will not cause the identifiers to be overwritten. Furthermore, *labelRules* takes a *String* prefix, which is added to the begin of the identifier. This way we can use the function to change the *Id*s of a set of rules. For example, we can modify the *Id*s of rules of a specific actor or a particular phase in the conversation. A disadvantage of using *labelRules* is that it does not make sure whether the number is an unique identifier. Using numbers for custom labels is discouraged, because they are more likely to interfere with this way of computing identifiers.

4.2.4 Strategies

A conversation is built up from phases. Each phase resembles a theme that is used in the training. A strategy for a single phase is a combination of the interactions from a client and a pharmacist. There are many ways to define a strategy for a single phase.

In Figure 4.12 we define a strategy, and two sub–strategies. The strategy starts with an introduction strategy, named *intro*. The *intro* and rest of the strategy are combined with the sequence combinator $<\star>$. The strategy always starts with the introduction, so we can use the guard of *intro* as a guard to start the phase. The rest of the strategy is an interleaved combination of two sub–strategies, using the $<\|>$ combinator as defined by Heeren et al.[36] Using interleave, we leave it up to the definition of the guards which part of a strategy is applicable when.

For the two sub–strategies, we define a list of rules, and apply the *exhaustive* function. The function *exhaustive* :: *IsStrategy* $f \Rightarrow [f \;\; a] \rightarrow$ *Strategy* a takes a list of strategies, and return a strategy that accepts all sequences of rules, as long these rules are applicable. Since we used guards to define when a rule is applicable and *exhaustive* only allows applicable rules, we can control which sequences are returned by *exhaustive* through guards. We define separate lists of rules for the agents, named *client* and *pharmacist*. By separating the rules per actor, we achieve an agent–like definition of rules.

Adding a new part of conversation is easily done by adding the new rules to the lists of the right phase. The higher–level strategy remains unchanged.

```
103 pharmacist :: Strategy Conversation
104 pharmacist = labelRules "pharmacist"
105                $ exhaustive [firstGoal
106                             , tellIntakeEffects
107                             , firstGoal2
108                             , ...]
109 client :: Strategy Conversation
110 client = labelRules "client"
111          $ exhaustive [clientv
112                       , clientt
113                       , ...]
114 strategy :: LabeledStrategy Conversation
115 strategy  = label "pharmacy.casus1.phase2"
116              $ labelRules "phase2"
117              $ intro <*> pharmacist <||> client
```

Figure 4.12: Defining the strategy in a phase of a conversation using interleaving.

Adapting interactions of the conversation only requires a change in the rules. Creating a similar client for the same pharmacist role is easy by reusing the pharmacist's strategy.

Figure 4.13 shows how to build up different phases of a conversation into the strategy for a whole conversation. In the example case we have five phases. We always start with the first phase, called *F1.strategy* in the code. The first phase can lead to the second or third, depending on the choices of the player. The second phase can lead to either the third, fourth or fifth phase. The case has three different endings to the conversations. Since the guarding of the phases *FEnd.strategyf4* and *FEnd.strategyf5* require *F2.strategy* to be passed, we chain the *FEnd.strategyf3* with *FEnd.strategyf4* and *FEnd.strategyf5* using the choice combinator. Instead of using choice for the second and third phase, we define the strategy by making the second phase optional and sequence it to the choice of an ending phase.

```
118 strategy1 :: LabeledStrategy Conversation
119 strategy1 = label "pharmacy.casus1"
120            $ labelRules "pharmacy.casus1"
121                $    F1.strategy
122               <*>  option F2.strategy
123               <*>  FEnd.strategyf3
124               <|>  FEnd.strategyf4
125               <|>  FEnd.strategyf5
```

Figure 4.13: Defining a strategy in phases.

Adding a new phase of conversation to a strategy is as easy as extending the current strategy by combining a phase. A new strategy *strategy2* can be combined with *strategy1* with a new *extension* by writing *strategy2* =

strategy1 <⋆> *extension*. Because the high–level definition of a strategy is clear, changing an existing strategy is not difficult.

4.3 Design Choices

During the development of the communication exercises in the framework, we had to make several design choices. Our implementation was based on a conversations script by a communications lecturer of the department of Pharmaceutical Sciences. Our goal was to find the least complex and most usable framework that would still be able to fully describe the script. We can choose how to implement the concepts of the term, the rules and strategy. When these three elements are defined carefully, defining the exercise is straight forward.

The next subsections explain which alternatives we explored when trying to implement a domain reasoner for communication in an educational setting. Sometimes comparing the alternatives shows that some choices are preferences instead of better solutions.

4.3.1 Usage of the Strategy language

We started from the point where a rule would consist of an utterance and effect on the feedback parameters. We defined a strategy with sequences of and choices between unguarded rules. Figure 4.14 shows pseudo–code for an example strategy that structures rules by <⋆> and <|> . The progression of interactions in a conversation is written out in full. Atomic strategies like *askUsedBefore* and *answerNotUsed* are the action and reaction by pharmacist and client. This approach doesn't separate the pharmacist's and client's interactions. It is a way of encoding a tree of interactions, similar to a dialog tree. With this approach, we would have to define the whole tree.

```
126 simpleStrategy  =  (askUsedBefore <⋆> answerNotUsed <⋆> ...)
127                  <|> (askTalkDoc <⋆> answerDocPrescribed <⋆> ...)
128                  <|> ...
```

Figure 4.14: Defining the strategy in a tree like manner.

Improving on the verbose tree–like definition of a strategy, we theorized a way of combining strategies cleverly such that it would decrease the size of the complete strategy. Wanting to resemble actual conversation, we looked for ways of combining strategies that resembled interactions for one actor. The first idea was to define a new combinator, which we called the 'lock–step' combinator. Lock–stepping two strategies a and b is denoted by a <?> b. A lock-step allows each sub–strategy to propose a step by taking turns. The problem with lock–stepping is that two sub–strategies always take turns, making it impossible for one of the two strategies to apply two rules. There was an example situation in which the client keeps on talking, and the pharmacist is to cut off the client at some point. We allow the client to proceed until the pharmacist decides to take over control of the conversation. To allow the definition of such a strategy, we need a more flexible way of allowing two strategies to take turns.

Using guards to control when a rule is applicable, is a flexible way of defining strategies. Rules are extended with a guarding function such that we can compute if a rule is applicable. Using the *exhaustive* combinator function, which takes a list to execute rules until there is no–one that is applicable, we could create concise strategies. An efficient alternative is to build a list of action–reaction pairs, as shown in Figure 4.15. These action–reaction pairs use the characteristic of our conversation that a client usually answers to a pharmacist. This solution would lessen the amount of work on defining guards and modifiers, since we can treat two rules as one. If all interactions are part of a pair, we reduce the work on defining guards and modifying functions with fifty percent.

```
129 phase = exhaustive [askName         <*> giveName
130                   , cutToTheChase   <*> walkAway
131                   , giveMedicine    <*> endConversation]
```

Figure 4.15: Code of an example strategy, in action–reaction pairs.

Because we can't assume that an NPC only has one reaction to an action of the player, we interleave two lists of guarded rules. For example, consider a situation where a pharmacist explains a lot about the medicine. A client might interrupt somewhere, to ask a question. In that case, we would want to use more than one interaction to define the informative talk of the pharmacist. Interleaving the strategies a and b, is denoted by $a <\|> b$. Interleaving two strategies that use *exhaustive* allows us to separate rules towards actor–specific sub–strategies. Figure 4.16 shows interleaving of the example used earlier to demonstrate pairs of interactions.

```
132 phase  =  exhaustive [askName
133                      , cutToTheChase
134                      , giveMedicine]   -- pharmacist
135        <||> exhaustive [giveName
136                      , walkAway
137                      , endConversation]   -- client
```

Figure 4.16: An example strategy of interleaved, agent–specific sub–strategies.

This way we could work towards the definition of a pharmacist strategy that could possibly communicate with any client it faces. As a consequence we have to invest more work in defining the rules, since we need to guard and modify the state for each rule. Compared to the list of pairs, interleaving sub–strategies is more adaptable since we can easily add or remove rules. We could define a strategy such that it uses reaction pairs where possible. Moreover, it allows for an interleaved strategy for sub–strategies of agents.

4.3.2 Defining Sentences

Before using the IDEAS framework, we tried to recreate the core components to see what would be desirable. We started with a definition of a rule that

only consisted of the utterance as a string, as in Figure 4.17. The constructor had the type *Rule* :: *String* → *Rule*. A strategy like *pharmacist* in the example controls when a rule was offered in an option for the player. With two strategy definitions interleaved, it is possible to simulate conversation. A conversation without options was not enough to serve as exercises for an educational game. We needed more information to create multiple choice questions from the strategy.

```
138 qMedication = Rule "What do you know of metformin?"
139 aMedication = Rule "I was hoping you could tell me."
140 pharmacist = (qMedication <|> aMedication) <⋆> ...
```

Figure 4.17: A first definition of rules and strategy.

Next, we used different constructors for every rule to capture the basic intention of a sentence. The constructors in Figure 4.18 can be used for generating feedback according to the type of an interaction. Giving feedback that states that an answer should follow after a question is not so valuable.

```
141 data Interaction where Question, Answer, Concern, Inform,
142                        Action, Interrupt :: String → Interaction
```

Figure 4.18: Specific constructors for *Interaction*s allow to differentiate interactions on the intention of a sentence.

We need to generate specific feedback for an educational game. Giving feedback on the added types was not specific enough. We improved the feedback by decorating every rule with a function that builds up feedback parameters. The modifiers on client parameters and learning goals was the result. Based on the values of the parameters we can compare rules and paths of a scenario. Secondly, we can add detailed feedback labels to specific sub–strategies. We give detailed and localized feedback to the learner based on rule comparisons and labels.

4.3.3 State Safety

Currently the conversation knowledge is a mapping from *String* to *Int*, but it is possible to use any type as keys. It is pragmatic to use *String*s because adding a mapping is as easy as adding a pair of a String and an *Int*. The problem is that using *String*s for keys is error prone. Figure 4.19 shows that a mistake is easily made when *String*s are used to look up values. Using constant strings as keys is considered a good programming practice, but still that is not the perfect solution. If we have two different constants that have the same value, using these constants as keys of a mapping will conflict when used differently.

What we would like, is that the keys are typed, like in Figure 4.20. The compiler offers the security that every key used is actually an applicable key in the mapping. We can use an enumeration, because of its small universe of values. Using *SafeKeys* would mean that we can only map values to keys that

```
143    > import Data.Map
144    > let newMap = Data.Map.insert "some-string" 1 empty
145    > Data.Map.lookup "some_string" newMap
146 Nothing
147    > Data.Map.lookup "some-string" newMap
148 Just 1
```

Figure 4.19: An illustration of an easily made mistake, when using *String*s as keys, in a GHCI session.

reside in the enumeration. If we were to use a key in a lookup that was not part of *SafeKeys*, the compiler returns an error, that such a data constructor is not in scope. If we were to utilize the typed value as keys, the compiler gives us the guarantee that all guarding–functions and conversational knowledge modifiers are type–safe. Using *SafeKeys* has as disadvantage that the keys are not extendible without changing the code. Using new keys, requires the data type *SafeKeys* to be changed.

```
149 data SafeKeys = SomeKey
150                | AnotherKey
151                | AndSoOn
152   deriving (Show, Eq, Ord)
153 type SafeMap a = Map SafeKeys a
154   -- example usage of a SafeMap
155 exMap :: SafeMap Int
156 exMap = m
157   where m' = insert SomeKey 1 empty
158         m  = insert AnotherKey 0 m'
159   -- Data.Map.lookup SomeKey exMap => Just 1
160   -- Data.Map.lookup AndSoOn exMap => Nothing
161   -- Data.Map.lookup Unknown exMap => error
```

Figure 4.20: Example of a *Map* with type–checked key values.

Still, trying to lookup the value of a mapping that is not inserted will return *Nothing*. It is unwanted to define a guard or modify function that uses a lookup of a value that is not inserted. We could define an algorithm that checks the required values in modify and guard functions and those that are inserted. Although it is not possible for an algorithm to fix such an error, it could return warning messages for a scenario editor. Warnings point to possibly erroneous parts of a scenario.

A *Conversation* that consists of maps using type–safe keys can prevent some errors by a scenario writer. Defining a type that can provide keys for the number of values, required for the scenario will require extra effort by the scenario writer. It is a trade–off between effort and potential errors.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The communication training of pharmacy students consists of conversation theory and practice. The students are taught models to conduct conversations in a structure manner. Models help structure a phases of a conversation in the correct order. Furthermore, the course highlights conversation skills such as asking open questions. During the practice, students perform a conversation together and give feedback afterwards. The interactions are defined by their utterance, place in conversation and the effect on the context.

According to the functional and technical requirements we compared three software frameworks that describe conversation and generate feedback. We found two frameworks that have been used to describe communication: a dialog tree and BDI agents. Because domain reasoners have been used for describing exercises, we developed a reasoner for the communications domain. The domain reasoner scores best compared to the other two. The dialog tree is simple in structure, but does not offer the feedback generating functionality of a BDI model or a domain reasoner. The framework that builds a Domain reasoner is less complex than a framework for BDI agents, and the extra expressiveness that a BDI model offers, is not used in our specific case. We have no need for two intelligent agents because our client always reacts the same to a pharmacist, and a pharmacist agent should give options to let the player choose.

Using an implementation of rules that rewrite the state in the IDEAS framework, allows us to define strategies that solve an exercise in the form of a conversation. Strategies are combinations of rewrite rules such that the state changes during the conversations. Rewrite rules are interactions with a guard and function that modifies feedback parameters. We use the guarding functionality of rewrite rules to define when rules are applicable. The information of the guards allows us to use the applicable rules as guidance within a phase. We can define sequences of and alternative phases to create the high–level structure of conversation. The functions that modify the feedback parameters build up feedback during a conversation. These functions can be used to compare rules and sub-strategies on the parameter values.

By using the IDEAS framework for developing a communication domain reasoner, the framework shows its capabilities outside mathematical, logical and

programming domains. For now it is unclear what domains might be formalised in the framework in the future, but we expect that the framework could be applied to more domains in the future.

5.2 Future Work

The following sections discuss more expressive usages of the IDEAS framework. Furthermore, we predict that different subjects of the conversations could potentially lead to different functional requirements. Lastly, we argue that an exercise editor aides writers of scenarios and would lead to less errors during the development of a scenario.

5.2.1 Context–dependent Rules

Until now we have only used specific sentences for an interaction. Figure 5.1 shows how two ordinary *String*s are used as sentences of interactions. These sentences contain context information, like the names of the clients, `"Miss Darcy"` and `"Mister Bond"`. We see possibilities in increasing the expressiveness of a rule by abstracting over the context information.

```
1   -- Specific
2 interaction, interaction2 :: String
3 interaction = "Miss Darcy, this is your medicine, Metformin 500."
4 interaction2 = "Mister Bond, this is your medicine, a Martini."
```

Figure 5.1: Context–specific definitions of two similar interaction.

Defining context–dependent rules hinges on the definition of a context. Figure 5.2 shows the definition of a simple data type that stores specific information. In the example, we define the name of the client and the drug they are using. In general we can store names of persons and items.

Instead of defining a rule using a single sentence, we define a sentence with variables for the case–specific information of the specific situation. For example, we can define two similar interactions in Figure 5.2. The regular rules, *interaction* and *interaction2* consist of a single *String*. The context-dependent definition *cfInteraction* defines a function that builds a *String* when given extra information from the context. The rule *cfInteraction* could be used in any strategy. The functions *name* and *drug* return the correct values from a *Context* which can be substituted to build the final interaction. We can define two specific interactions based on one rule and the correct context.

Currently, rules are not created such that they are dependent on context information. Even though it is always possible to abstract over a client's name for example, if you never need that abstraction, it seems to be no improvement compared to specific rules. Another reason for not defining context–dependent rules, is that it only proves to be more efficient when there are strategies with different contexts that feature the same interaction. If there is no way of telling which interaction can be reused in other strategies, how can we recognize a potential context–dependent rule?

```
5  data Context = C {
6     drug :: String,
7     name :: String
8     }
9  msDarcy = C {name = "Miss Darcy", drug = "Metformin 500"}
10 mrBond  = C {name = "Mister Bond", drug = "a Martini"}
11    -- Dependent
12 cfYourmedicine :: Context → String
13 cfYourmedicine c = ((name c) ++ ", this is your medicine, "
14                      ++ (drug c) ++ ".")
```

Figure 5.2: Defining a *Context* for client information enables the context–dependent definition of interactions.

Maybe the greatest advantage of context-dependent rules is the possibility of achieving a single strategy for the pharmacist. If rules are not client specific, we could define a strategy for the pharmacist, that could handle any client it faces. Moreover, by combining strategies for different conversations, we achieve a strategy for the pharmacist that can handle multiple patients. Since every conversation starts with an introduction phase, in which the client is introduced, we could build up the context information using the state modifiers. Ideally, we could factor out the client's specific details such that the specific information could be provided by the client's strategy. The usefulness of context–dependent rules should be looked into. We see opportunities for context-dependent rules, but they cannot replace, and should coexist with specific rules.

5.2.2 Extended guards and Modifiers

Also for future research, we imagine guards and modifier functions that have different meanings during the strategy. We see possibilities for rules to be reused, but with slightly different guarding and effect. The pragmatic solution is to say that similar rules are all defined as distinct rules. But another approach is to define one rule, that changes throughout the strategy.

For example, in our scenario we can choose to address a client with her name. It is not strange to use a name once, but after a couple of times it becomes awkward. The positive effect on the parameter of making contact decreases by using interactions that address the client by name. We could define a set of rules with similar interaction but different functions for guarding and modifying. A rule's modifier could use the number of times the name is used in interactions to have different effect. The effect on the contact parameter can be calculated depending on the specific situation.

Lets assume that the first time a pharmacist addresses the client by name, the rewarded is higher than in following interactions. Furthermore, the effect turns negative when the player uses the name too much. A modifying function that changes on an integer value could be defined as in Figure 5.3. If we have not used the name before the i value would be 0. The first time of using an interaction with the *contactWithName* modifier would result in 2 points on the scale of making contact. The following two times the modifier would result in 1

```
1 contactWithName :: Num a ⇒ Int → (LearningGoals a)
2 contactWithName i | i ⩽ 0      = idLearningGoals { contact = 2}
3                   | i > 3      = idLearningGoals { contact = − 1}
4                   | otherwise = idLearningGoals { contact = 1}
```

Figure 5.3: Example modifier for making contact by addressing the client by name.

point for making contact. After three times the effect drops to -1, resulting in a decrease for the making contact.

In the example, the number of times the player could use an interaction with a name is arbitrarily chosen. It is only to be expected that these numbers could vary, depending on the client. If we were to have a *Context*, similar to the one in the previous section, we would want to define a context-dependent variable for these integer values.

The definition of a rule that depends on an integer is very straightforward. Using such a rule requires the possibility to count the number of interactions that have used the client's name. The pragmatic way is to keep a counter in the state and use a modifying function to update the counter accordingly. Another way would be to analyse the conversation to calculate i. The framework keeps track of the rules that have passed, called a prefix. Currently rules cannot access the prefix. To allow rules access to the current visited path requires a change in the framework.

Another example is a rule that is applicable when the client is agitated. If the client is agitated, the pharmacist can address the client on its agitation. Currently, we define when a client is agitated and if that is the case, we offer the player the option to address that. If we guard a rule on the client parameters, we could implement a strategy that reacts to client. We could define strategy that can react to a client that got agitated before the conversation started. The difficulty of an exercise could be changed by changing the client parameters.

It is interesting to see if the extended functionality can be used efficiently. It is difficult to see how a scenario progresses, when you have both specific rules, that follow after each other, and state– and context–dependent rules that change during the conversation, or are dependent on the starting values. The readability and maintainability of a scenario might be decreased when defined with regular rules and state– and context–dependent rules. Using high–level rules allows the writer to write more concise rules. We should look into the usefulness high–level rules.

5.2.3 Introducing Knowledge System

During the project we struggled with the use of a domain specific knowledge in the conversation. We wanted to give feedback and hints based on the clients background and medication. A knowledge system for pharmaceutical knowledge allows a computer to reason about health. For example, if the knowledge base can deduct a side–effect of the given medicine for pregnant women, we can deduce the applicability of questions towards pregnancy. Using information about the client, like its gender and age, we would want to address the implications of

the medication and maybe even suggest alternatives. A knowledge base could allow defining rules that depend on situational and domain specific knowledge. Using the information gives a realistic background to exercises. The conversation is tailored to the client's specific situation Implementing such system to work together with the framework would require a change in how the framework finds rules. The framework should not only check the guarding functions, but also use the knowledge base when finding applicable rules.

5.2.4 Analysis of Similar Communication Domains

In this research we have analysed communication training as it was used at the department of Pharmacy of Utrecht University. Although we assume that a similar structure is found when analysing other communication training for masters, it has yet to be proven that training exercises of other courses, can be defined by the current framework. The serious communication–training game as a result of the project Communicate, will be prototyped by students of the department of Pharmacy. The next steps in developing an all-round communication game involve the analysis of different kind of conversations. We expect that changes will be made to the client parameters and learning goals. The mapping used for describing the conversation knowledge is probably flexible enough to be used for other conversations

5.2.5 Exercise editor

An exercise editor can aid the writer of scenarios in a communication domain. The conversation domain reasoner differs from the mathematical, programming and algebra domain reasoners. There are three reasons why the communication domain would benefit from an editor, that are not applicable to the previous use cases of the IDEAS framework.

First of all, scenarios require a large number of rewrite rules. The framework was developed with a mathematical definition of rules in mind. In mathematics, there is often one, or maybe a few ways to solve a problem with a set of rewrite rules. Mathematical exercises often reuse the same strategy to solve similar exercises. In communication, every conversation requires its own interactions. Although context-dependent rules can offer a higher level of reusing rules, we will still need to define more rules than for some other domains. Compared to the mathematical domain, we have a large set of rewrite rules in the conversation domain. An exercise editor would help the writer to create a larger set of rules.

Secondly, one of the characteristics of mathematical rules is that once they are defined, they are either correct or buggy. A buggy rule is used to describe a common mistake, that allows the definition of detailed feedback. When the definition of rules is not so strict, in communications for example, interactions often need a process refinement to ensure its quality. The process of refinement requires small adjustments while the scenarios are tested. While refining rules, we need to update the framework. We could either define a function that allows the recompilation of rules and strategies, or we could allow the exercise editor to recompile the framework, and update the web–server accordingly.

Lastly, an editing application can create the pre–conditions in the state and validate the scenario. Mathematical rules have a clear set of rewrite rules, that transform one equation to another equation. Interactions in conversations

modify a state, which is not insightful like an equation. It consists of variables and parameters. An editor would allow the writer of the scenario to focus on the actual conversation and defining the right feedback, without worrying to much about modify functions for variables and parameters.

5.2.6 Validation

Given a rule's guarding function, we can validate that the rule is applicable at some point in the strategy. Furthermore, we could validate modifier functions. Modifier functions are only necessary when they rewrite the state, such that they enable some paths in the strategy or lead to the generation of feedback.

Secondly, we could define functions that validates a strategy. Such a function can find sub–strategies that are never offered as an option to the player. Duplicate rules can be detected automatically. Furthermore, we could check a strategy for cycles.

Validations could be implemented as functions of a meta service[1] of the framework. Moreover, a scenario editing application could apply these functions to give warnings to the writer. What all validation functions add to an editor, is insight in the sometimes complex domains and their rewrite rules. It is difficult for a scenario writer without programming knowledge, to keep overview of all paths in the strategy when looking at the rules and strategy definitions. Adding automated checks to a structured view on the exercise aid the development of exercises.

[1] `http://hackage.haskell.org/package/ideas-1.1/docs/src/Ideas-Service-ServiceList.html#metaServiceList`

Bibliography

[1] A. J. Stapleton, "Serious games: Serious opportunities," in *Australian Game Developers' Conference, Academic Summit, Melbourne*, 2004.

[2] D. Charsky, "From edutainment to serious games: A change in the use of game characteristics," *Games and Culture*, vol. 5, no. 2, pp. 177–198, 2010.

[3] T. Susi, M. Johannesson, and P. Backlund, "Serious games: An overview," 2007.

[4] I. Erev, A. Luria, and A. Erev, "On the effect of immediate feedback," in *Y. Eshet-Alkalai, Y., Caspi, A. & Yair, Y.(Eds.), Learning in the Technological Era. Proceedings of the Chais Conference, March*, vol. 1, pp. 26–30, 2006.

[5] A. Gerdes, B. Heeren, J. Jeuring, and S. Stuurman, "Feedback services for exercise assistants," in *The Proceedings of the 7th European Conference on e-Learning*, pp. 402–410, 2008.

[6] J. Jeuring, H. Passier, and S. Stuurman, "A generic framework for developing exercise assistants," in *Proceedings of the 8th International Conference on Information Technology Based Higher Education and Training, ITHET*, 2007.

[7] J. Lodder, J. Jeuring, and H. Passier, "An interactive tool for manipulating logical formulae," in *Proceedings of the Second International Congress on Tools for Teaching Logic, Salamanca, Spain, September 26 - 30, 2006* (B. P. L. M. Manzano and A. Gil, eds.), 2006. Also available as Technical report Utrecht University UU-CS-2006-040.

[8] J. Jeuring, A. Gerdes, and B. Heeren, "Ask-Elle: A Haskell tutor," in *21st Century Learning for 21st Century Skills* (A. Ravenscroft, S. Lindstaedt, C. Kloos, and D. HernÃąndez-Leo, eds.), vol. 7563 of *Lecture Notes in Computer Science*, pp. 453–458, Springer Berlin Heidelberg, 2012.

[9] L. Blom, M. Wolters, M. ten Hoor-Suykerbuyk, J. van Paassen, and A. van Oyen, "Pharmaceutical education in patient counseling: 20h spread over 6 years?," *Patient Education and Counseling*, vol. 83, no. 3, pp. 465–471, 2011.

[10] A. Zande, "KNMP-standaarden voor zelfzorg," *Pharmaceutisch Weekblad*, vol. 130, no. 5, pp. 108–109, 1995.

[11] M. Wooldridge and N. Jennings, "Agent theories, architectures, and languages: A survey," in *Intelligent Agents* (M. Wooldridge and N. Jennings, eds.), vol. 890 of *Lecture Notes in Computer Science*, pp. 1–39, Springer Berlin Heidelberg, 1995.

[12] K. Van den Bosch, A. Brandenburgh, T. J. Muller, and A. Heuvelink, "Characters with personality!," in *Intelligent Virtual Agents*, pp. 426–439, Springer, 2012.

[13] A. Schaafstal, "Knowledge and strategies in diagnostic skill," *Ergonomics*, vol. 36, no. 11, pp. 1305–1316, 1993. PMID: 8262025.

[14] N. Schwarz, "Cognition and communication: Judgmental biases, research methods, and the logic of conversation," *Ann Arbor*, vol. 1001, pp. 48106–1248, 1996.

[15] K. Scalise and B. Gifford, "Computer-based assessment in e-learning: A framework for constructing "intermediate constraint" questions and tasks for technology platforms," *The Journal of Technology, Learning, and Assessment*, vol. 4, June 2006.

[16] P. Botella, X. Burgués, J. Carvallo, X. Franch, G. Grau, J. Marco, and C. Quer, "ISO/IEC 9126 in practice: what do we need to know?," in *Proceedings of the 1st Software Measurement European Forum*, 2004.

[17] N. Bevan, "Quality in use: Meeting user needs for quality," *Journal of Systems and Software*, vol. 49, no. 1, pp. 89 – 96, 1999.

[18] J. Bøegh, "A new standard for quality requirements.," *IEEE Software*, vol. 25, no. 2, pp. 57–63, 2008.

[19] A. Abran, A. Khelifi, W. Suryn, and A. Seffah, "Usability meanings and interpretations in ISO Standards," *Software Quality Journal*, vol. 11, no. 4, pp. 325–338, 2003.

[20] Wikipedia, "Dialog tree — Wikipedia, the free encyclopedia." `http://en.wikipedia.org/w/index.php?title=Dialog_tree&oldid=553516989`, 2013. [Online; accessed 29-November-2013].

[21] B. Ellison, "Defining dialogue systems." `http://www.gamasutra.com/view/feature/132116/defining_dialogue_systems.php`, 2008. [Online; accessed 13-December-2013].

[22] R. Rosenfeld, D. Olsen, and A. Rudnicky, "Universal speech interfaces," *interactions*, vol. 8, pp. 34–44, Oct. 2001.

[23] A. S. Rao, M. P. Georgeff, *et al.*, "BDI agents: From theory to practice.," in *ICMAS*, vol. 95, pp. 312–319, 1995.

[24] T. J. Muller, A. Heuvelink, K. van den Bosch, I. Swartjes, *et al.*, "Glengarry glen ross: Using BDI for sales game dialogues.," in *AIIDE*, 2012.

[25] A. Solimando and R. Traverso, "Designing and implementing a framework for BDI-style communicating agents in Haskell," in *Declarative Agent Languages and Technologies X*, pp. 203–207, Springer, 2013.

[26] M. Sulzmann and E. S. Lam, "Specifying and controlling agents in Haskell,"

[27] B. Heeren, J. Jeuring, and A. Gerdes, "Specifying rewrite strategies for interactive exercises," *Mathematics in Computer Science*, vol. 3, no. 3, pp. 349–370, 2010.

[28] P. Hudak *et al.*, "Building domain-specific embedded languages," *ACM computing surveys*, vol. 28, no. 4es, p. 196, 1996.

[29] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Comput. Surv.*, vol. 37, pp. 316–344, Dec. 2005.

[30] B. Heeren, J. Jeuring, A. Van Leeuwen, and A. Gerdes, "Specifying strategies for exercises," *Intelligent Computer Mathematics*, pp. 430–445, 2008.

[31] B. Heeren, J. Jeuring, and A. Gerdes, "Properties of exercise strategies," *First International Workshop on Strategies in Rewriting, Proving, and Programming*, pp. 21–34, 2010.

[32] B. Heeren and J. Jeuring, "Adapting mathematical domain reasoners," in *Intelligent Computer Mathematics* (S. Autexier, J. Calmet, D. Delahaye, P. Ion, L. Rideau, R. Rioboo, and A. Sexton, eds.), vol. 6167 of *Lecture Notes in Computer Science*, pp. 315–330, Springer Berlin Heidelberg, 2010.

[33] J. Jeuring and B. Heeren, "Ideas: interactive domain reasoners," 2010.

[34] A. Gerdes, *Ask-Elle: a Haskell Tutor.* PhD thesis, Open Universiteit Nederland, 2012.

[35] B. Heeren, A. Gerdes, and J. Jeuring, "Hackage: ideas: Feedback services for intelligent tutoring systems," 2013.

[36] B. Heeren and J. Jeuring, "Interleaving strategies," in *Intelligent Computer Mathematics*, pp. 196–211, Springer, 2011.