Incorporating BDI agents into human-agent decision making research

Bart Kamphorst, Arlette van Wissen, and Virginia Dignum

Institute of Information and Computing Sciences, Utrecht University, the Netherlands

Abstract. Artificial agents, people, institutes and societies all have the ability to make decisions. Decision making as a research area therefore involves a broad spectrum of sciences, ranging from Artificial Intelligence to economics to psychology. The Colored Trails (CT) framework is designed to aid researchers in all fields in examining decision making processes. It is developed both to study interaction between multiple actors (humans or software agents) in a dynamic environment, and to study and model the decision making of these actors. However, agents in the current implementation of CT lack the explanatory power to help understand the reasoning processes involved in decision making. The BDI paradigm that has been proposed in the agent research area to describe rational agents, enables the specification of agents that reason in abstract concepts such as beliefs, goals, plans and events. In this paper, we present CTAPL: an extension to CT that allows BDI software agents that are written in the practical agent programming language 2APL to reason about and interact with a CT environment.

1 Introduction

Decision making has since long been an area of interest to scholars from all kinds of disciplines: psychology, sociology, economics and more recently, computer science. A lot of research has been done on finding, isolating and formalizing the factors that are involved in decision making processes of both humans and computer agents [18] [11] [22]. The Colored Trails (CT) framework [10] is designed to aid researchers in this purpose. It is developed (i) to study interaction between multiple actors (humans or software agents) in a dynamic environment and (ii) to study and model both human and agent decision making. CT allows for a broad range of different games to be implemented, such as one-shot take-it-or-leave-it negotiation games [7], iterated ultimatum games [27] and, of late, dynamic games with self-interested agents [26].

Currently, CT software agents are *computational agents* implemented in the object oriented programming language Java. In this paper, we will use the term 'computational agent' to refer to software agents that determine their strategy by use of algorithms, probabilities or game theory. We will use this term to distinguish these agents from agents that use concepts from folk psychology to define strategies. CT agents are usually tailored either to display one type of behavioral strategy, such as egoism or altruism [27], or to maximize their utility

for every action [16]. However, even though computational agents perform well in some scenarios, they lack the *explanatory power* to help understand the reasoning processes involved in decision making. Computational agents may make optimal decisions based on a clever probabilistic algorithm, but they will generally not show you how they did it.

In order to gain more insights into the actual reasoning processes that lie behind a decision of a software agent, the agents must be endowed with a richer model of reasoning. Based on Bratman's theory of rational actions in humans [2], agents can be constructed that reason in abstract concepts such as beliefs, goals, plans and events [3]. These types of agents are often referred to as Belief, Desire and Intention (BDI) agents. Ideally, agents are able to display *reactivity*, *proactiveness* and *social abilities* [32]. That is, they should be able to perceive and respond to the environment, take initiative in order to satisfy their goals and be capable to interact with other (possibly human) actors. For an agent to have an effective balance between proactive and reactive behavior, it should be able to reason about a changing environment and dynamically update its goals. Having social abilities requires it to respond to other agents, for example by cooperating, negotiating or sharing information. Working in a team for instance requires agents to plan, communicate and coordinate with each other. A BDI architecture lends itself well to implement these requirements in an intuitive yet formal way [9].

The BDI approach has proved valuable for the design of agents that operate in dynamic environments. It offers a higher level of abstraction by explicitly allowing beliefs to have a direct impact upon the agents behavior. This means the agents can respond flexibly to changing circumstances despite incomplete information about the state of the world and the agents in it [6]. Since BDI uses 'mental attitudes' such as beliefs and intentions, it resembles the kind of reasoning that we appear to use in our everyday lives [30]. To interact successfully, agents can benefit from modeling the mental state of their environment and their opponent [23]. Additionally, BDI models provide a clear functional decomposition with clear and retractable reasoning patterns. This can provide more helpful feedback and more explanatory power.

This paper presents middleware that lets software agents with a BDI decision structure interact with humans and other software agents in CT. The software presented in this article, CTAPL, allows BDI software agents that are written in the practical agent programming language 2APL to reason about and interact with a CT environment and the actors within the environment. CTAPL is a platform designed for the implementation of various interaction scenarios between BDI agents, computational agents, humans and heterogeneous groups. Although the framework of CTAPL has been developed, we are currently in the preliminary stages of evaluating CTAPL by building BDI agents whose performance can be compared to the performance of computational CT agents.

2 Related Work

The BDI model of agency does not prescribe a specific implementation [25]. We therefore do not claim that 2APL is the only suitable agent language for decision making research. There exist several different implementations that differ from each other in the logic they use and the technology they are based on. More often than not, the logics that are used are not formally specified in the semantics of the BDI programming language. 2APL differs from most agent languages (e.g., JACK [28] and Jadex [21]) in that it is defined with exact and formal semantics [3]. In addition to being theoretically well-motivated, 2APL provides easy implementation of external environments (see Section 4.1 for a more in-depth discussion of these environments), enabling different environments and frameworks to be connected to it.

The study of mixed-initiative interactions requires some kind of negotiation environment for the interactions to take place. Most environments are domain-specific and focus on specific tasks to be evaluated. Multi-agent decision making environments are mostly designed for agent simulation based on human performance. Examples of these environments are OMAR [5] and GENIUS [14]. The CT framework is very flexible in that it can be used to implement domains ranging from highly abstract to rich and complex scenarios. However, using BDI agents in decision making environments is certainly not limited to CT and it would be interesting to see how well BDI agents can be incorporated into other negotiation environments.

Literature shows that there are frameworks constructed for BDI agents [19] and recent work focuses on the development of a multi-agent simulation platform that supports different agent-oriented programming languages [1]. However, these frameworks are designed for very specific domains and are therefore not broadly applicable. CTAPL allows for the implementation of a range of different interaction domains in which BDI agents, computational agents and humans can interact. We are not aware of any other existing generic framework that allows computational agents and BDI agents to interact with each other and with humans in negotiation environments.

3 Colored Trails

Throughout the remainder of the paper we will illustrate various aspects of the CT framework using a decision making scenario that was presented in [27]. In this scenario the agents do not have to cope with any uncertainties about the world except the strategy of the opponent. Furthermore, it is a game that can be easily be divided into goals and subgoals. That is, players can create their own decision recipe trees that do not involve probabilities. This game is therefore very suitable to be implemented in CT as well as in the BDI constructs of 2APL. The game consists of an implementation of the ultimatum game (UG) [13], in which two players (here referred to as Alice and Bob) interact to divide their colored chips in order to take a path to the goal.

3.1 The framework

CT [10] is a flexible research platform developed by Grosz and Kraus to investigate decision-making in group contexts. CT implements a server-client architecture. Figure 1 represents the conceptual design of CT: A is the set of software agents and $\{A_1 \ldots A_i\} \in A$. H is the set of human actors and $\{H_1 \ldots H_i\} \in H$. Through an administrative shell a configuration file is loaded that specifies the properties of the game. Once the configuration file is loaded, the server starts the game.

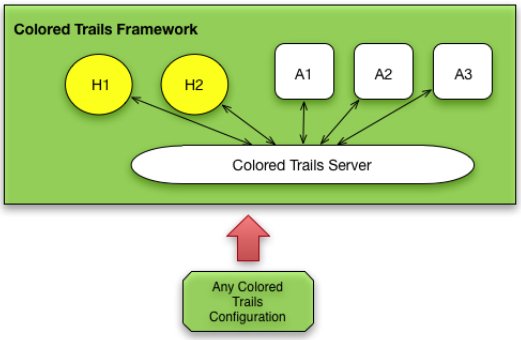

Fig. 1. The Colored Trails framework

The standard CT environment consists of a board of colored squares, one or more goals, two or more players (these can be both humans and software agents) and a set of colored chips per player. Players may move across the board by handing in colored chips that correspond to the colored squares of their taken path. They are allowed to negotiate in order to obtain useful chips. The size of the board, the colors of the squares, the number of players and the number of goals are a few examples of the many variables that can be set in the configuration file. The configuration file also specifies when and in what way players may communicate with each other and what information each player has about the current state of the world. CT thus allows for games of both *imperfect* and *incomplete information*. CT also allows for the "specification of different reward structures, enabling examination of such trade-offs as the importance of the performance of others or the group as a whole to the outcome of an individual and the cost-benefits of collaboration-supporting actions"[10]. Given the large number of variables that can be modified, a great variety of domains can be implemented in CT.

Figure 2 shows the configuration of the ultimatum game that was mentioned earlier. The board consists of 5×5 colored squares and two players, Alice and Bob, who each have full visibility of the board and of the other player's position, goal and chips. This means they do not have to cope with any uncertainty, other

than the strategy of the opponent. On the board, the position of both players and their goal is visible. The chips (displayed in the 'Player Chips Display' at the bottom of the screen) represent the resources the players have and can divide amongst themselves. A player is either a proposer and required to propose the split of the chips, or a responder and required to respond to an offered proposal.

3.2 What's missing?

The CT framework was specifically designed to investigate human-agent interactions: "A key determinant of CT design was the goal of providing a vehicle for comparing the decision-making strategies people deploy when they interact with other people with those they deploy when computer systems are members of their groups."[10] If a scenario focuses on analyzing decision making in interactions between humans and agents, it can be interesting and helpful if the agents reason in a similar way as humans (say they) do. These agents help us understand how they interact and what motivates their decisions, by enabling us to look more closely at their reasoning process. On top of that, agents that are based on models that take into account the same social principles that humans also base their decision on (such as fairness and helpfulness), were shown to explore new negotiation opportunities [15] and find solutions that correspond to solutions found by humans [4]. The information about the agent's reasoning process can be used to create agents that are able to support humans in decision making tasks.

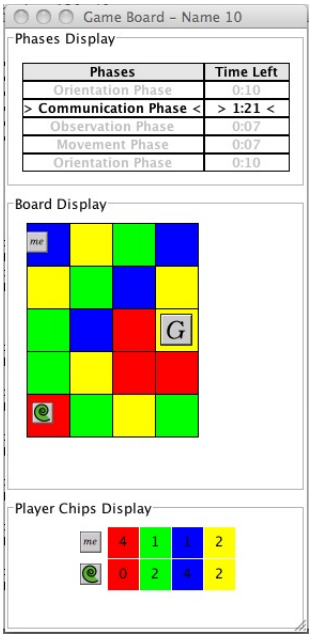

Fig. 2. The Configuration of a Colored Trails Game

BDI structures reflect the collaborative decision making process of humans more closely and at a more realistic level than the more algorithmic approaches [29]. The possibility of having BDI agents within the CT framework also allows for comparisons between how people interact with BDI agents and with computational agents. Currently, the CT framework is only suitable for computational agents written in Java. What is missing from CT is a way to have BDI agents interact in CT domains.

4 2APL

4.1 The platform

2APL (pronounced double-a-p-l) is a practical agent programming language designed to implement agents that can explicitly reason with mental concepts such as beliefs, goals (desires) and plans (intentions) [3]. Figure 3 shows the conceptual design of 2APL. Like CT, the 2APL platform implements a server-client architecture, where A is the set of BDI agents written in the 2APL programming language and $\{A_1 \ldots A_i\} \in A$. Each agent A_n can communicate with all other agents via a *send action*. All communication between agents goes through the 2APL server.

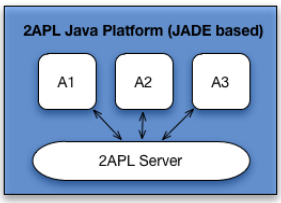

Fig. 3. The 2APL platform

The 2APL language has both a formal syntax and a formal semantics, which makes it possible to check programs on the satisfiability of their (formal) specifications and constraints. Furthermore, it integrates both declarative and imperative programming styles. Goals and beliefs are stated in a declarative way, while plans and external environments are implemented in an imperative programming style. The declarative programming is well-suited for the implementation of reasoning and the updating of the mental states of the agents. The imperative part enables the implementation of plans, flow of control and allows for the interaction with existing imperative programming languages. (For a more detailed discussion of the implementation of beliefs, goals and plans in 2APL, see [3].) The 2APL platform also allows for *external environments*. These environments are modular extensions that agents can have access to through *external actions*. External environments serve as an interface to the Java programming language which allows programmers (i) to build custom environments for agents to interact in and (ii) to easily add functionality to the 2APL Platform.

Example 1 illustrates a specific reasoning pattern of the UG example in 2APL. Alice might have a different strategy of reacting to proposals received from Bob than Bob has of reacting to Alice's proposals. It might be the case that Alice is egoistic and only accepts proposals if it leaves herself better but Bob worse off, while Bob is more altruistic and also accepts proposals that favor Alice. In 2APL this can be expressed in terms of goals, subgoals and belief updates.

This example shows what happens if Alice receives a proposal. The method 'makeResponse(MSGID)' is called when Alice receives a new proposal and she has an ID. If Alice has the goal to win and to respond to proposals (meaning that she is currently the responder in the game), then she will check whether accepting an offer that Bob has proposed is both beneficial for her and harmful for Bob. If this condition is met, she will accept the proposal and put this in her belief base. Otherwise, she will reject the proposal and put this in her belief base. Statements that transfer knowledge to the belief base can be identified by the first letter of the name of the method, which is always a capitol. In this case, the statement 'Responded(TYPE, MSGID, accept)' is a belief update rule (as defined in 2APL), which will update the belief base with the fact that the agent responded with 'accept' to a particular proposal. Since in the following round Alice will be a proposer, she now updates her goal base by dropping the subgoal of responding to proposals and accepting the goal of making proposals.

Example 1 (Performing high level task-specific reasoning: $makeResponse(MSGID)$)

```
makeResponse(MSGID) <- received(TYPE,MSGID,open) and agentId(
    MYID) | {
if ( G(win) and G(respond) ) then {
      if ( B(scoreAfterExchange(MYID,SCORE) >
          scoreCurrentChips(MYID))
         and B(scoreAfterExchange(BOBID,SCORE) <
         scoreCurrentChips(BOBID) ) then {
            response(MSGID,accept);
            Responded(TYPE,MSGID,accept)
      } else {
            response(MSGID,reject);
            Responded(TYPE,MSGID,reject)
      }
      dropGoal(respond);
      adopta(propose)
   }
}
```

4.2 What's missing?

Although the 2APL platform in principle allows external environments to have a graphical user interface (GUI) with which humans can interact with the 2APL agents, external environments are in practice mostly designed to examine the agents' behavior and reasoning processes. The environments provided by 2APL are not very well-suited to study human-agent interaction, because the scenario often focuses on helping the agent to learn or display certain behavior. However, since BDI systems have the advantage that they use similar concepts of

reasoning as humans do, it would be very interesting to study their behavior in heterogeneous settings comprising of both agents and humans. This requires a empirical testbed that enables the implementation of both abstract and more real-world domains in which humans and agents can interact. CT is very suitable for this purpose.

In many scenarios, the BDI model has proven to be a useful tool and several successful applications use BDI structures [29] [20]. According to Georgeff, "the basic components of a system designed for a dynamic, uncertain world should include some representation of Beliefs, Desires, Intentions and Plans [...]" [8]. However, one of the main criticisms against BDI systems is that it cannot deal properly with learning and adaptive behavior. However, recent attempts have been made to extend BDI languages with learning components [12]. Extensions to the existing BDI framework can be easily evaluated in CT, since agents have to adapt to a dynamic environment and can learn from interactions with humans. "The CT architecture allows games to be played by groups comprising people, computer agents, or heterogenous mixes of people and computers. [...] As a result, CT may also be used to investigate learning and adaptation of computer decision-making strategies in both human and computer-agent settings" [10]. Another criticism concerns the gap between the powerful BDI logics and practical systems [17].

To conclude, the 2APL platform is currently missing a uniform way to let humans interact with the BDI agents. Combining 2APL and CT enables researchers to study BDI agents in a setting of human-agent interaction.

5 CTAPL

From the previous sections, two issues can be distilled. The first is that the CT framework in its current state lacks a clear-cut way to build agents with a rich model of reasoning needed to help better understand the reasoning processes involved in decision making. Secondly, although the 2APL platform offers a BDI agent programming language that provides agents with such a rich model of reasoning, the 2APL platform is in itself not very suitable for human-agent interaction experiments. CTAPL is designed to overcome both problems. CTAPL is middleware that allows BDI agents, written in 2APL, to participate in a CT environment in which heterogeneous actors can interact. CTAPL allows for both the interaction between BDI agents and humans and between BDI agents and computational agents. It is even possible to have a mixed group of BDI agents, computational agents and humans interact. Because of this, CTAPL is a very suitable platform to evaluate the performance of BDI agents.

5.1 Conceptual Design

Figure 4 shows the conceptual design of CTAPL. The top layer represents the 2APL platform. As with any 2APL setup, it consists of a server and one or more BDI agents ($A_1 \ldots A_i$). The bottom layer represents the CT framework,

with software agents and human actors. In CTAPL however, the CT agents are not fully functional, reasoning agents. Instead, $GA_1 \ldots GA_i$ are mere hooks for the BDI agents to communicate with the CT environment. Each agent A_n thus corresponds with hook GA_n. The 2APL platform is extended by an external environment that instantiates (i) the hooks for each 2APL agent and (ii) a Java Thread that continually listens whether agents have received any new messages from the server. In CTAPL the communication between agents flows through the CT messaging system instead of directly through the 2APL server.

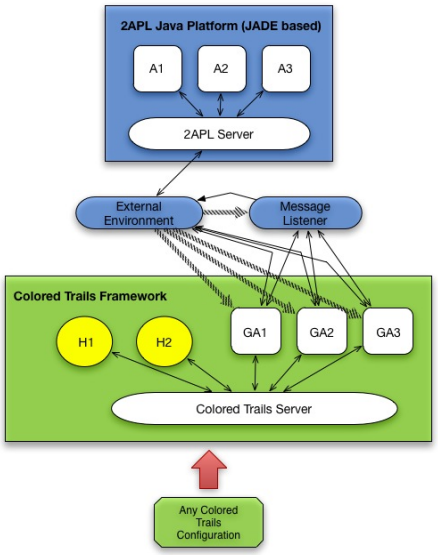

Fig. 4. The conceptual design of CTAPL

5.2 Implementation

CTAPL consists of four major components: a 2APL external environment, a MessageListener class, a set of hooks in CT in the form of generic Java CT agents and higher level 2APL code that 2APL agents have to use in order to communicate with the CT framework. In the upcoming subsections the four components will be discussed individually.

2APL Agents The 2APL agents are designed to perform all higher level reasoning about the game. Through external actions the agents can pass information to and request information from the CT environment. CTAPL provides a 2APL API that defines BDI constructs for all external actions that are available in the CTAPL external environment (see Section 5.2). For instance, the

procedural rule[1] *getPosition(ID)*, shown in Example 2, allows agents to retrieve the position of another actor on the board. This method can be called without restrictions, as demonstrated by the guard of this method, which is 'true'. First, the method calls upon the 'getPosition(ID)' method in the CT environment. The environment returns the result of this call, which is captured in the variable POS. Subsequently, it is checked whether the POS value meets the required form of two coordinates. These coordinates are then put in the belief base with the belief update rule 'Position(ID,X,Y)'.

Example 2 (Requesting information from the external environment: $getPosition(ID)$**)**

```
getPosition(ID) <- true | {
    @ctenv(getPosition(ID), POS);
    B(POS=[X,Y]);
    Position(ID,X,Y)
}
```

The agent programmer can include the constructs from the API by putting an *Include: filename* statement on the first line of the 2APL file that defines the reasoning patterns of the specific BDI agent. This statement creates a union between the code from the API and the agent code designed by the agent programmer. Other extensions such as general beliefs and plans specified by the programmer may also be put into separate files that can then be included by all agents.

The External Environment The external environment consists of several classes, written in Java with the 2APL API, that have been packaged as a Java archive (jar). External environments are the default way in the 2APL platform of providing agents with an environment in which to interact. Normally, an external environment is a closed system that defines all the external actions an agent can perform within the environment. In CTAPL however, the external environment functions as a proxy between the 2APL agents and the hooks in CT. For instance, the environment ensures that the procedural rule *getPosition(ID)* from Example 2 will call the *getPosition(ID)* method of the CT hook that corresponds with the 2APL agent in question and returns the value to the 2APL agent. If Alice – programmed in 2APL – wants to know Bob's position, the 2APL code will call the CT environment to extract this information.

CT Generic Agents The generic agent functions solely as a hook for 2APL to communicate with the CT server. The class *GenericAgentImpl* contains all the basic functionality that CT offers its regular agents. Examples of basic methods are *getChips(), getPosition()* and *setClientName(String name)*. Example 3

[1] For more on procedural rules in 2APL, see [3].

shows the getPosition(ID) method in CT, to give an impression of the functionality. First, the set of players is collected by the method 'getPlayers()', which is defined in the CT API. The method then cycles through the set of all players to find the one with the right ID. Then the position of this player is collected by 'getPosition()', which is also defined in the CT source code. The method now stores this position value and returns it to the environment, where the method was called.

Because the agent functions as a hook to the 2APL platform, GenericAgentImpl also implements higher level methods such as *getBoard(), getPlayers()* and *getPlayerByPerGameId(int pergameid)*. If CT is extended by new methods specifically designed for a specific experiment, the generic agent class can be subclassed to implement the additional methods.

Example 3 (The CT functionality: *getPosition(ID)*)

```
public RowCol getPosition(int id) {
    Set<PlayerStatus> players = client.getPlayers();
    RowCol position = null;
    for (PlayerStatus player : players ) {
        if (player.getPerGameId() == id) {
                    // the getPosition() method is defined in
                        the CT API
                    position = player.getPosition();
                    break;
         }
    }
    return position;
}
```

The Message Listener The Message Listener component is a Java Thread instantiated by the external environment that continuously polls each CT Generic Agent for new messages that the agent may have received.[2] This information is then passed through the external environment onto the 2APL agents using the 2APL built-in construct *throwEvent(APLFunction event, String ... receivers)*. This construction allows the BDI agents to passively gain knowledge about messages that have been sent to them. The 2APL agent code specifies what to do when messages are received.

[2] Technically, the Message Listener is part of the external environment. However, because it is a Threaded class and serves a very specific purpose, it is considered a separate component of CTAPL.

6 Discussion

Currently, the authors are in the process of implementing an egoistic BDI agent in 2APL that interacts within a scenario similar to the one described in [27]. A proof of concept has already been developed that shows that the framework of CTAPL functions properly by relaying information from 2APL to CT and visa versa. Once the implementation of the egoistic BDI agent is complete, the authors will evaluate it by comparing its performance with that of the egoistic agent from [27]. We expect the BDI agent to perform *at least* as well as the original agent.

CT allows for environments in which uncertainties, probabilities and utilities play an important role. Due to the highly abstract philosophical origins of BDI, such concepts are typically not included in BDI models. One might therefore object against the use of 2APL in CT by raising the question how 2APL would handle such concepts. In defense of CTAPL two things may be said. The first is that this objection holds for almost all approaches that use BDI structures. The question posed brings forth a long-existing tension between those who favor BDI models and those who favor computational (decision and game theoretic) approaches to building software agents [31]. The burden of solving this tension does not lay with CTAPL because it is written as an extension to the already existing framework of CT. The authors of this paper do not claim that BDI agents should be preferred over computational agents in all cases. Instead, we have argued that BDI agents can be a valuable addition to decision making research with CT when the focus is on understanding the reasoning processes involved.

Second, the 2APL platform does offer a clear, modular way to capture uncertainties, probabilities and utilities. External environments may be written that provide computational methods. By allowing BDI agents access to these environments, the agents can request a certain value to be calculated. The agents can then reason with the outcome. Consider the case in which Alice does not know what chips Bob has. Alice may now access an external environment to calculate the probability of Bob having the chips Alice needs. Alice can then use this information to make her offer. So even though the probability is calculated using a computational algorithm, Alice uses her own beliefs, desires and intentions to interpret that value. In this way the 2APL agents are able to reason about uncertainties, probabilities and utilities.

7 Future Work

Currently no work has been done to deploy BDI agents in CT scenarios in which agents have to deal with (i) utilities and (ii) uncertainties about the world. As we discussed in section 6 traditional BDI systems are not well equipped to cope with uncertainties. Current BDI approaches simply do not provide tools for quantitative performance analysis under uncertainty. Future research will have to show whether these difficulties can be overcome by either using external

environments to implement computational features that BDI agents can utilize, or by using hybrid approaches such as BDI-POMDP to deal with uncertainty [24].

As mentioned in section 6, the authors will use CTAPL to compare the performance of an egoistic BDI agent with that of an egoistic computational agent in a negotiation scenario similar to the one presented in [27]. Because CTAPL allows for the interaction between both types of agents, the platform is very suitable for empirically comparing BDI agents with computational agents in terms of speed, performance and explanatory power in various settings. Future work will provide more empirical insights into the advantages and disadvantages of the BDI approach. Future research with CTAPL also includes (i) building BDI agents that model human decision making processes in a setting of coalition formation with self-interested agents [26] and (ii) improving the planning mechanism of agents in a collaborative setting with uncertainty [16].

8 Conclusions

In this paper the authors have proposed a technical solution for dealing with the explanatory gap that exists when computational agents are used to investigate decision making. We have proposed that BDI based agents can assist in filling the gap because they use clear and retractable reasoning patterns. This paper has described new middleware called CTAPL that is designed to combine the strengths of a BDI-based agent approach with the Colored Trails testbed for decision making.

CTAPL makes three major contributions. First, CTAPL opens up the possibility for BDI researchers to explore existing research domains developed in CT for agent-agent interaction. Secondly, it gives BDI researchers the opportunity to have BDI agents that perform optimally in a certain environment interact with human players. Lastly, it creates the possibility for CT researchers to write agents that can qualitatively reason in terms of beliefs, goals and plans by using the 2APL agent programming language.

Acknowledgements. We thank Ya'akov (Kobi) Gal and Maarten Engelen for helpful comments and assistance with the initial setup of CTAPL. This research is funded by the Netherlands Organization for Scientic Research (NWO), through Veni-grant 639.021.509.

References

1. R. Bordini et al. Mas-soc: a social simulation platform based on agent-oriented programming. *Journal of Artificial Societies and Social Simulation*, 8(3), 2005.
2. M. Bratman. *Intentions, Plans and Practical Reason.* Harvard University Press, 1987.
3. M. Dastani. 2apl: a practical agent programming language. *Autonomous agents and multi-agent systems*, 16(3):214–248, 2008.
4. S. de Jong, K. Tuyls, and K. Verbeeck. Fairness in multi-agent systems. *The Knowledge Engineering Review*, 2008.

5. S. Deutsch and M. Adams. The operator-model architecture and its psychological framework. In *6th IFAC Symposium on Man-Machine Systems*, MIT, Cambridge, MA, 1993.
6. F. Dignum, D. Morley, E. Sonenberg, and L. Cavedon. Towards socially sophisticated bdi agents. In *Proceedings of the Fourth International Conference on Multi-Agent Systems*, Boston, MA, 2000.
7. Y. Gal and A. Pfeffer. Predicting people's bidding behavior in negotiation. *AAMAS*, 2006.
8. M. Georgeff, B. Pell, M. Pollack, and M. Tambe. The belief-desire-intention model of agency. *Lecture Notes in Computer Science*, Jan 1999.
9. B. Grosz and S. Kraus. Collaborative plans for complex group action. *Artificial Intelligence*, 86(3):269–357, 1996.
10. B. Grosz, S. Kraus, S. Talman, B. Stossel, and M. Havlin. The influence of social dependencies on decision-making: Initial investigations with a new game. *AAMAS*, 2004.
11. B. Grosz, A. Pfeffer, S. Shieber, and A. Allain. The influence of task contexts on the decision-making of humans and computers. *Proceedings of the Sixth International and Interdisciplinary Conference on Modeling and Using Context*, 2007.
12. A. Guerra-Hernández, A. E. Fallah-Seghrouchni, and H. Soldano. Learning in bdi multi-agent systems. In *Computational Logic in Multi-Agent Systems*. Springer Berlin / Heidelberg, 2005.
13. W. Guth et al. An experimental analysis of ultimatum bargaining. *Journal of Economic Behavior and Organization*, 3:367–388, 1982.
14. K. Hindriks, C. Jonker, S. Kraus, R. Lin, and D. Tykhonov. Genius - negotiation environment for heterogeneous agents. Budapest, Hungary, May 2009.
15. L. Hogg and N. Jennings. Socially intelligent reasoning for autonomous agents. *IEEE Trans on Systems, Man and Cybernatics - Part A*, pages 381–399, 2001.
16. E. Kamar, Y. Gal, and B. Grosz. Incorporating helpful behavior into collaborative planning. *AAMAS*, 2009.
17. M. Mora, J. Lopes, R. Viccari, and H. Coelho. Bdi models and systems: Reducing the gap. In *LNCS: Intelligent Agents V: Agents Theories, Architectures, and Languages*, pages 11–27. Springer Berlin / Heidelberg, 2007.
18. A. Newell and H. A. Simon. *Human Problem Solving*. Prentice-Hall, 1972.
19. M. Nguyen and W. Wobcke. A flexible framework for sharedplans. In A. Sattar and B. Kang, editors, *AI 2006: Advances in Artificial Intelligence*. Springer Berlin / Heidelberg, 2006.
20. R. Onken and A. Walsdorf. Assistant systems for vehicle guidance: Cognitive man-machine cooperation. *Aerospace Science Technology*, 5:511–520, 2001.
21. A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A bdi reasoning engine. In *Multi-Agent Programming: Languages, Platforms and Applications*. Kluwer, 2005.
22. A. Sanfey, J. Rilling, J. Aronson, L. Nystrom, and J. Cohen. The neural basis of economic decision-making in the ultimatum game. *Science*, (300):1755–1758, 2003.
23. M. P. Sindlar, M. Dastani, F. Dignum, and J.-J. C. Meyer. Mental state abduction of bdi-based agents. In *Lecture Notes in Computer Science*, volume 5397, pages 161–178. Springer Berlin / Heidelberg, 2009.
24. M. Tambe et al. Conflicts in teamwork: Hybrids to the rescue. *AAMAS*, pages 3–11, 2005.
25. W. van der Hoek and M. Wooldridge. Towards a logic of rational agency. *Logic Journal of the IGPL*, 11(2):135–159, 2003.
26. A. van Wissen, B. Kamphorst, Y. Gal, and V. Dignum. Coalition formation between self-interested heterogeneous agents. *Forthcoming*.
27. A. van Wissen, J. van Diggelen, and V. Dignum. The effects of cooperative agent behavior on human cooperativeness. *AAMAS*, 2009.
28. M. Winikoff. Jack^{TM} intelligent agents: An industrial strength platform. In *Multi-Agent Programming: Languages, Platforms and Applications*. Kluwer, 2005.
29. S. Wolfe, M. Sierhuis, and P. Jarvis. To bdi or not to bdi, design choices in an agent-based traffic flow management simulation. *Proceedings of the 2008 Spring Simulation Multiconference*, 2008.
30. M. Wooldridge. Intelligent agents. In G. Weiss, editor, *Multiagent Systems*. MIT Press, 1999.
31. M. Wooldridge. *Reasoning about Rational Agents*. MIT Press, 2000.
32. M. Wooldridge and N. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.