ANALYSING HUMAN-AGENT TEAMWORK

Rafael H. Bordini[(1)], **Michael Fisher**[(2)], and **Maarten Sierhuis**[(3,4)]

(1)	*Dept. Computer Science, University of Durham, UK*	R.Bordini@durham.ac.uk
(2)	*Dept. Computer Science, University of Liverpool, UK*	MFisher@liverpool.ac.uk
(3)	*RIACS, NASA Ames, USA*	msierhuis@mail.arc.nasa.gov
(4)	*Dept. Mediamatics, Delft University of Technology, NL*	

Abstract We here address the modelling and analysis of human-agent teamwork, specifically in the context of proposed astronaut-robot collaboration in future space missions. We are particularly interested in modelling such systems at a level that allows formal verification techniques to be applied, and hence carry out sophisticated analysis of the reliability and effectiveness of the teams before the system is deployed in real scenarios. In this paper we describe our ongoing research in this area.

1 INTRODUCTION

Collaboration between humans and agents, particularly robotic agents, is increasingly important. Not only do high-profile activities such as exploration missions to the Moon and Mars by space agencies such as NASA and ESA intend to employ human-agent teams, but our everyday activities involving interaction with hardware devices and interaction through the *Internet*, especially in the context of pervasive computing, also fall into this category. In many of these scenarios, we are expected to *trust* that the agents will do what we expect and that the agents and humans will work together as expected. But: how can we be sure, or at least be more confident? In this paper, we bring together work on the verification of multi-agent systems with work on the modelling of human-agent teamwork to provide the first steps towards a solution. This paper proposes the use of the *Brahms* modelling framework, together with abstraction into *Jason* agent code, and application of automated agent verification techniques [1] in order to analyse such collaborative activities. The particular application area we consider is the analysis of human-*robot* teams intended for use in future space exploration missions. Missions into space are not only increasingly important, but represent applications where autonomy is central and where reliability and safety is vital. Missions to other planets clearly require some form of autonomy due to problems with communication. From a financial point of view, missions involving astronauts can be very expensive. Thus, spacecraft are increasingly controlled by software, leading to key examples where autonomy is required, such as the Remote Agent from Deep-Space 1 [16] and the Demonstration of Autonomous Rendezvous Technology (DART) [8].

An important concept in capturing the notion of autonomy is that on a *software agent*[1] [24]. An agent makes its own decisions about what activities to pursue. Beyond simple autonomy, *rational agents* are increasingly used as a high-level abstraction/metaphor for building of complex/autonomous space systems. Rational agents can be seen as agents that make their decisions in a rational and explainable way [25]. Thus, the key new aspects that such agents bring is the need to consider, when designing or analysing them, not just what they do but *why* they do it. Since agents are autonomous, understanding why an agent chooses a particular course of action is essential. The rational agent notion allows us to abstract away from low-level control aspects and to concentrate on the key features of autonomy, namely the goals the agent has and the choices it makes. Thus, in modelling a system in terms of rational agents, we typically describe each agent's *beliefs* and *goals*, which in turn determine the agent's *intentions*. In fact, the predominant approach in this area is that of using beliefs, desires, and intentions (BDI), where desires are a form of long term goal [15, 14]. Such agents then make *rational* decisions about what action to perform, given their beliefs and goals/intentions: they behave in a rational way that can be analyzed and explained.

2 MODELLING HUMAN-AGENT TEAMWORK

The use of the rational agent metaphor, particularly the representation of behaviour using *mental* notions such as beliefs, goals and intentions, has several benefits. This approach abstracts away from low-level control issues: we describe some goal that the agent wishes to be achieved, and continue without worrying about the detail of goal implementation. Perhaps more importantly, the view of participants (be they agents or people) in a system in terms of these mental notions is both intuitive and easier for humans to understand and predict. Not surprisingly, the modelling of complex systems, including space exploration systems, in terms of rational agents has been very successful, particularly through the *Brahms* framework [19, 18].

[1] In the rest of this paper we use the term *agent* wherever we mean *software agent*

2.1 Typical Teamwork Scenario

Below, we describe a particular human-agent teamwork scenario, taken from a Mars exploration context. It concerns the description of astronauts exploring the surface of Mars together with robots as a member of the team.

Basic Description. Two surface astronauts are going on an Extra-Vehicular Activity (EVA) to explore a region, defined by the crew in an EVA plan. A critical constraint is that both astronauts always have to have a network connection back to their habitat. To create such a network connection one or more network relays provide connectivity from the habitat to each astronaut. A network relay can be an EVA Robotic Assistant (ERA) or a Robotically-Deployed Relay Device (RDR). Whereas a RDR has only one function during the mission, i.e. that of a network relay, the ERA robots can be autonomous network relays, as well as astronaut assistants, carrying tools and sample bags, taking pictures and panoramas as well as collecting rock samples. Besides these two astronaut support functions, the ERAs can have their own defined autonomous exploration plan. These plans can include site surveying activities, including imaging, streaming video and sample collection activities that the ERA performs autonomously. Just as people, the ERAs can be interrupted by others when performing their exploration plan. When this occurs the robots need to handle such interruptions gracefully and without jeopardizing the safety of both astronauts and themselves.

Each astronaut can "team up" with a robot. When a robot is teamed up with an astronaut, the robot's personal agent (PA) automatically performs the "astronaut following" and "astronaut watching" activities[2], and also automatically begins to monitor network connectivity to the astronaut. To satisfy its function as a network relay, the astronaut's PA begins to monitor network connectivity from the habitat to the robot as well. The PA notifies the astronaut when network connectivity to the habitat fails. Using a defined model of teamwork the robot and astronaut PAs will provide recovery steps to try to re-establish communication.

The EVA plan assigns regions to be explored by the astronauts and by the ERA robots. However, whenever an astronaut (or another robot, under certain circumstances) needs help and requests that the robot performs a different task, the robot has to immediately interrupt what it is currently doing and take up the new activity, which can mean that the PA has new goals, in order to obtain new overall states that help resolve whatever problem the astronaut has. Most likely, such problem-solving activity means collaboration between the team of robots and astronauts. Needless to say, when the problem is resolved and the robot's assistance is no longer required, the robot can resume its exploration activity at the point it was previously interrupted.

In summary, we have a situation where autonomous robots will have to choose the best course of action in order to perform both its own exploration activities and its tasks for being the astronaut's robotic assistant. This means that the robot's PA has to appropriately perform both its planned activities in order to achieve the exploration goals initially assigned to it in the original EVA plan, as well as supporting its human-robot team tasks. To accomplish both, besides acting on these "long-term" exploration goals — or exploration activities in *Brahms* terms — whenever possible, the ERAs need to react to unpredictable events, such as possible loss of connectivity, as well as requests for help from other astronauts and robots. These new tasks also need autonomous decision on the part of the robots in regards to the best courses of action given the beliefs the ERA PAs have about the current state of the mission and the environment.

Key Behaviours, The details of a particular situation in the scenario above are given below (see Fig. 1).

- The astronaut has a PA (Astro PA; implemented in *Brahms*/*Jason*) to communicate with the robot agent.
- In simulation mode, the astronaut is modelled as a (*Brahms*/*Jason*) agent.
- There are two robots, `R1` and `R2`, each controlled by a (*Brahms*/*Jason*) agent (ERA PA).
- Initially `R1` is teamed up with the astronaut and together they form a human-robot team.
- `R1`'s agent is monitoring the network link back to the habitat and to the astronaut's agent.
- Astro PA is monitoring the network link to the robot.
- Astronaut is executing their part of an EVA plan.
- Robot is executing its part of the EVA plan.
- The astronaut is working on a stationary activity from its EVA plan near the robot.
- `R1` is working on take-panorama activity of the EVA plan.
- The astronaut tells his PA that he will start a move activity to a new location.
- The Astro PA tells the robot agent `R1` that the astronaut is moving to a new location.
- Because the astronaut is `R1`'s team member, the robot interrupts its activity of taking the panorama and follows the astronaut.
- At a point in time, `R1` loses connection with the habitat, and will inform the astronaut that it has to go back to the location where it last had connection. Because the astronaut is moving to a new location away from the robot, `R1` will ask `R2` if it can become the astronaut's team member, thus following the astronaut and keeping network connectivity for the astronaut to the habitat via `R1`.

[2] Astronaut following and watching means that the robot camera is used to follow the astronaut and streams video back to the habitat

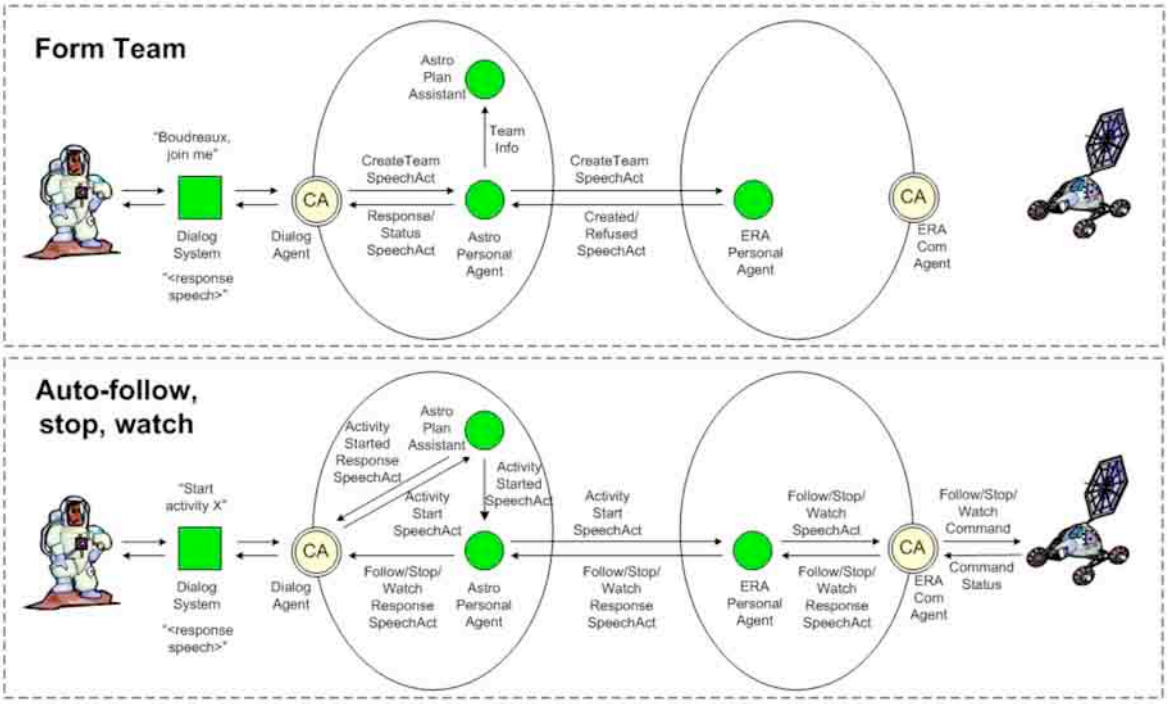

Figure 1: Human-Robot Teamwork Model

- The astronaut confirms that this is OK, and the robot moves back to its previous connectivity location.
- When `R1` comes back to its previous location, it connects to `R2` and asks that robot to become the astronaut's team member and move towards the astronaut's location, interrupting its current activity.
- `R2` tells `R1` that it is busy doing something, and asks if `R1` can wait.
- `R1` tells `R2` that it has to do it, because `R1` has lost connection to the habitat, moved back and as a consequence there is no connection to the astronaut at the moment.
- `R2` confirms it understands, interrupts its current activity and moves to the last known astronaut's location, trying to recover connection to the astronaut on the way.
- `R2` reaches the astronaut, becomes a team member, and follows the astronaut while maintaining the network connection.
- When the astronaut is done with its activity, `R2` tells the astronaut that it needs to go back and finish its previously interrupted activity.
- Together, `R2` and the astronaut move back to the robot's previous activity location.
- `R2` releases itself from the team and continues its previously interrupted activity.
- The astronaut is done with its EVA plan and goes back to the habitat while the robots finish their respective plans by themselves.

2.2 Brahms

Brahms is a multi-agent modeling and simulation and MAS development environment developed at NASA Ames Research Center [19]. Specifically, Brahms is a modelling language designed to model human activity. Agents, therefore, were developed to represent people. Brahms agents can belong to one or more groups, inheriting attributes, initial beliefs, and activities, workframes and thoughtframes from multiple groups (multiple inheritance). This allows the abstraction of agent behaviour into one or more groups. Because Brahms was developed in order to represent people's activities in real-world contexts, Brahms also allows the representation of artifacts, data, and concepts in the form of classes and objects. Both agents and objects can be located in a model of the world (the geography model) giving agents the ability to detect objects and other agents in the world and have beliefs about the objects. Agents can move from one location in the world to another by executing a *move* activity, simulating the movement of people. For a more detailed description of the Brahms language we refer the reader to [17] and [18].

Brahms agents are BDI-like agents. However, Brahms does not make use of the "goal" abstraction, but rather uses an approach we refer to as *activity-based* [5, 20]. Brahms agents are both deliberative and reactive. Brahms demonstrates how a multi-agent BDI language, symbolic cognitive modeling, traditional business process modeling, activity- and situated cognition theories are brought together in a coherent approach for analysis and design of organizations and human-robot systems.

We have developed Brahms as a tool for a software engineering methodology we refer to as "from multi-agent simulation to implementation" of socio-technical systems. Part of this methodology is to first model and simulate the current operations of a group and, using the output metrics of the simulation, to decide if and what type of automation might be useful to aid the group's work practice. Next, we use Brahms to design, model and simulate how this automation will be integrated within the current operation, effectively creating a simulation of the future operations of the group. The last step is to turn the model of the designed system into software agents that can be implemented and executed in actual operations.

The Brahms environment consists of different tools to develop Brahms models, simulate and execute Brahms models and display agent and object interactions of multi-agent models, as well as an event database to, for example, generate agent performance statistics. Brahms is freely available for research purposes at the Brahms project website (`http://www.agentisolutions.com`). The Brahms language was designed to model people's work practice, as opposed to an organization's work process [17]. The concept of work practice is derived from the social sciences, in particular, business anthropology. The study and observation of people in an organization leads to insights into the practices of the people. It is these practices that implement organizational procedures and processes. The Brahms language was developed to model organizations at the practice level by representing each individual's activities as they are performed within a situational context of people, places, systems, individual, group and organizational practices. To avoid misconceptions about the often ill-defined concept of work practice, we first define what is meant with this. *Work practice* is the collective performance of contextually situated activities of a group of people who coordinate, cooperate and collaborate while performing these activities synchronously or asynchronously, making use of knowledge previously gained through experiences in performing similar activities.

Thus, the Brahms language was originally conceived of as a language for modeling contextual situated activity behavior of groups of people. This created two very important ideas for the language: first, to model a group of people it is very natural to model them as software agents; second, modeling contextual situated behavior of a group imposes a constraint on the level of detail that is useful in modeling the dependent and independent behavior of the individuals. The right level is a representational level that falls between functional process models and individual cognitive models [6]. If we are interested in modeling a *day-in-the-life* of say ten or more people, modeling the individual behavior at the level of cognitive task models will be very time consuming, because these models are generally at the millisecond decision-making level. To overcome this kind of detail, the Brahms language uses a more abstract level of behavioral modeling that is derived from Activity Theory [23, 12] and Situated Action [20]. An individual's behavior is represented in terms of activities that take an amount of discrete time and can be decomposed into more detailed sub-activities if necessary.

The Brahms language is a pure agent-oriented programming language. It is *not* a set of Java libraries enabling agent-based programming in the Java language. Instead, Brahms is a full-fledged multi-agent language allowing the modeler to easily and naturally represent *multiple agents*.

Brahms Descriptions. The Brahms language is organised around the following representational constructs:

Groups of groups containing

→ Agents who are located and have
→ *Beliefs* that lead them to engage in
→ *Activities* specified by Workframes

Workframes in turn consist of

→ *Preconditions* of beliefs that lead to
→ *Actions*, consisting of
→ *Communication* Actions
→ *Movement* Actions
→ *Primitive* Actions
→ Other composite activities
→ *Consequences* of new beliefs and facts
→ *Thoughtframes* that consist of
→ *Preconditions* and *Consequences*

Thus, one can see that the underlying concepts within Brahms fit into the BDI model [14]. Indeed, a full formal semantics, were one to exist, would no doubt make use of such concepts. However, as such formal semantics does not exist, we must tackle verification in a different way, as we discuss in Section 4.

Brahms Example. The Brahms source code below shows the definition of the ERA group (line 1). In the group there is a situation-action rule (workframe) to start the ERA's autonomous plan (line 21 to 32). When this workframe is executed, the ERA first start the Relay function (line 29). Next, the ERA will execute the robot's plan (line 30). Line 35 and 36 shows the definition of the two ERA agents `R1` and `R2`.

```
1  group ERA memberof Robot  {
2    activities:
3       composite_activity startRelayFunction(PersonalAgent robotPA) {
4          ...    /* implementation omitted */
5       }//startRelayFunction
6
7       composite_activity executePlan(int pri, Plan plan, PersonalAgent robotPA,
8                                      Robot robot)
9       {
10         priority: pri;
11         end_condition: detectable;
12         detectables:
13         detectable detectPlanFinished {
14             detect((current.currentPlanFinished = true))
15             then end_activity;
16         }//detectPlanFinished
17         ...  /* the rest of the implementation is omitted */
18      }//executePlan
19
20     workframes:
21         workframe wf_startAutonomousPlan {
22            repeat: true;
23            when
24               ( knownval(current.isWaitingForInput = false) and
25                  knownval(current.isComingHome = false) and
26                  knownval(current.currentActivityStarted = false) and
27                  knownval(current.currentActivityInProgress = false) )
28            do {
29               startRelayFunction(robotPA);
30               executePlan(1, oPlan, oRobotPersonalAgent, oRobot);
31            }//do
32        }//wf_startAutonomousPlan
33   }//group ERA
34
35   agent R1 member of ERA { }
36   agent R2 member of ERA { }
```

2.3 *Jason*

In the area of autonomous agents and multi-agent systems, AgentSpeak is one of the best known agent-oriented programming languages based on the BDI architecture. It is a logic-based agent-oriented programming language introduced by Rao [13], and subsequently extended and formalised in a series of papers by Bordini, Hübner, and colleagues. Various *ad hoc* implementations of BDI-based (or "goal-directed") systems exist, but one important characteristic of AgentSpeak is its theoretical foundation and formal semantics. *Jason* is a Java-based platform for the development of multi-agent systems. At the core of the platform lies an interpreter for an extended version of AgentSpeak. We do not have the space here to introduce the language; however, a detailed account of the *Jason* language and platform can be found in [4]. The platform is available *open source* under GNU LGPL at `http://jason.sf.net`.

Practical BDI agents are implemented as reactive planning systems: they run continuously, reacting to events (such as perceived changes in the environment and new goals to achieve) by executing plans given by the programmer. Plans are courses of *actions* that agents commit to execute so as achieve their goals; actions are things the agents are able to do in order to change the environment, such as a robot moving its location. The pro-active behaviour of agents is possible through the notion of goals (desired states of the world) that are also part of the language in which plans are written.

An important aspect of an AgentSpeak interpreter is a data structure called the *set of intentions*. Intentions represent particular courses of action that the agent already committed itself to perform. Different intentions in this set represent different foci of attention for the agent, for example, a robot may need to concomitantly handle: responding to communication from an astronaut's PA; react to loss of communication with the habitat; as well as keeping track of the long-term goals assigned to it in the EVA plan. A partly executed plan will also typically have further goals the agent should adopt. The autonomous choice of which course of action to choose in order to achieve those goals is left as late as possible, so that the agent can use its most up-to-date information about the state of the environment in order to choose the best course of action. Of course, in very dynamic environments, plans can still fail to achieve the goal they were meant to achieve. That is why *Jason* features a plan-failure handling mechanism.

Jason has many features which we cannot cover here due to lack of space, but one of its main characteristics is that it is

highly customisable. It also has a language extensibility mechanism called *internal actions* which allows, for example, arbitrary Java code to be used from within an agent's reasoning cycle. This mechanism was also used to provide a library of useful internal actions for programmers. These are pre-defined internal actions distributed as part of the platform which allow, among other more practical things, complex manipulation of the BDI structures of the *Jason* interpreter. Of relevance here are the internal actions which allows programmers to suspend intentions and later resume them. This is important, for example, if an activity needs to be interrupted, as in the case of an astronaut requiring immediate action of a robot currently in the middle of its part of an EVA plan.

Just to give a taste of what the language look like consider the following examples of plans that could be used to model the scenario in Section 2.1.

```
1   ...
2
3   +!photo(Target,Zoom,Time)
4     :  not full(photo_archive)
5     <- ?position(Target,Coordinates);
6        point_camera(Coordinates);
7        camlib.set(Zoom);
8        shoot(Time);
9        ?photo(Target,Zoom,Time);
10       camlib.check_last_shot.
11
12  -!photo(Target,Zoom,Time)
13    :  not full(photo_archive)
14    <- !photo(Target,Zoom,Time).
15
16  ...
17
18  +!teamed_up_with(Astronaut)[source(Agent)]
19    :  .desire(photo(_,_,_)) | .desire(panorama(_,_,_))
20    <- .send(Agent,tell,busy);
21       +rejected_goal(teamed_up_with(Astro)[source(Agent)]).
22
23  ...
```

A robot's part of an EVA plan is likely to require the robot to traverse to some far location, then take a photo of a particular rock on that location, and so forth. The code of the agent controlling a robot would involve plans for moving around, as well as all other tasks a robot is expected to execute. The daily tasks assigned in a particular EVA plan would be communicated to the agent in the form of an AgentSpeak plan. The plan would make references to the plans for moving around and at some point requiring the agent to take the a photo of a particular target. This is the first example we gave above. (In AgentSpeak, an exclamation mark denotes an "achievement goal" and a question mark denotes a "test goal"; the example below should make clear what they are used for.)

The plan starting on line 3 says that *whenever* the agent comes to have a new goal of achieving a state of affairs in which a `photo` is believed to have been taken of a target represented by the logical variable `Target` with a particular `Zoom`, *provided* (line 4) it is **not** the case that the agent believes that its `photo_archive` is already `full`, a possible course of action the agent could adopt is as follows: (line 5) retrieve from the belief base what the agent currently believes to be the `Coordinates` of the `Target`'s `position`; (6) execute an action to move the robot's camera so that it's pointing to those coordinates; (7) execute a user-defined *internal* action available in their `camlib` library (which presumably would modify the internal settings to adjust the zoom); (8) execute the action to take the shot; (9) the code in this line is used to check if the attempt at taking the photo was successful (if not, there will be no matching belief in the belief base and the test goal fails); and finally the agent could use some image processing (presumably implemented in Java or some other imperative language) in order to check, e.g., if the focus of the photo was of good quality.

If any of those actions or goals fail, the plan in line 12 would be used. It essentially just instructs the agent to try again to achieve the goal of taking the photo, provided the reason for the failure was not that the archive was full (presumably because there would be other specific plans for that case). There is also an example of a plan involving inter agent communication (starting on line 18). It says that whenever an `Agent` wants the agent executing this code to achieve the goal of teaming up with a particular `Astronaut`, if the agent is currently taking a photo or panorama (a more detailed reading of the code is: it has the `desire` to achieve a state of affairs in which it believes that an arbitrary `photo` or `panorama` has been taken), it could `send` a message `tell`ing that `Agent` that it is `busy`, and then making a mental note (adding a belief to itself) so as to remember that this request was initially rejected (assuming that is necessary). Of course, there would be specific plans to react to urgent requests, as the one in the reply of `R1` to `R2` in the scenario in Section 2.1.

3 ANALYSING BEHAVIOUR USING FORMAL VERIFICATION

3.1 Formal Verification

The approach we advocate for ensuring efficiency and safety of team-work for space exploration is *formal verification*. In this work, the behavioural requirements we have of complex systems can be specified using formulae from an appropriate formal logic. Importantly, the formal logic used can incorporate a wide range of concepts matching the view of the system being described, for example *time* for dynamic/evolving systems, *probability* for systems incorporating uncertainty, *goals* for autonomous systems, etc. This gives great flexibility in the descriptive language that can be used. Given such a specification, we can check this against models/views of the system under consideration in a number of ways. The most popular is that of *model checking* [7]. In such approach, the specification is checked against all possible executions of the system; if there is a finite number of such executions, then this check can often be carried out automatically. Indeed, the verification, via model checking, of both hardware systems (such as chip designs) and software systems (such as Microsoft device drivers) has been very successful in industry as well as academia.

It is important to remember that, in verifying complex systems, such as those controlling autonomous space activity, the intention is not to analyse every low-level aspect, such as propulsion systems, etc. These lower-level electronic/mechanical aspects are analysed in other ways, and we abstract away from these details by giving a high-level logical specification of their behaviour [21]. Importantly, not only can we specify what the *expected* behaviour of such underlying systems are, but we can analyse how the overall system behaviour is affected *if* such systems do not perform as expected. Essentially, we aim to describe the overall expected behaviour of the controlling software in a logical framework appropriate for the properties we are interested in.

3.2 Formal Verification of Agents

We have been developing techniques for verifying multi-agent systems [1, 2, 3]. The agents involved are rational (and, so, are capable of "intelligent" autonomous behaviour) and are represented in a high-level language describing their beliefs, intentions, etc. In particular, we have looked at the verification of BDI languages [9], including AgentSpeak/*Jason* [13, 4]. In order to carry out formal verification, formal descriptions of both the system being considered and the properties to be verified are required. Producing formal descriptions for multi-agent systems is well understood, with the BDI approach to rational agency being particularly popular [15, 14].

While agent verification has the potential to be used in analysing agents in critical applications, there is an increasing need to consider the verification of situations involving more human-agent teamwork. For example, NASA's new focus on doing human-robotic exploration of the Moon and Mars brings human "agents" into the picture — this might simplify some of the autonomy requirements, but clearly increases safety and certification concerns. Autonomous space software is hard enough to verify already: the environments in which it executes is uncertain and the systems involve tend to be complex. Adding a human element to space exploration scenarios will surely make this even more complex. The emphasis here is not on individual behaviours, but on *team* aspects, such as coordinating activities or cooperating to achieve a common goal.

3.3 Current Problems

A typical problem in model checking, particularly of concurrent systems (e.g., where various autonomous entities have independent behaviour as in the space exploration scenario in Section 2.1 is that of *state space explosion*. The model checker needs to have an in-memory model of possible states of the system, and the number of such states grows exponentially, for example, on the number of different autonomous entities being modelled. The good news is that there are many different "state-space reduction techniques" which help alleviate the problem, and there is a fairly large research community working on such techniques. However, it is still early days for model checking of multi-agent systems so, for example, some of such techniques do not apply (as yet) to the kinds of programming language we expect to use in programming multi-agent systems.

Whereas such languages help with the complexity of developing such systems — for example by allowing the modelling of agents at the right level of abstraction — very little work exists in using them for modelling realistic scenarios of human (or human-robot team) activities, with perhaps Brahms being the exception. On the other hand, Brahms has no formal semantics, which makes it more difficult to do model checking, in particular in defining how the modalities of the logic language used to write the system specification (i.e., the expected behaviour of the system) is to be interpreted in regards to states of a Brahms model. This is essentially the main problem we are tackling in this joint work. In the next section, we briefly outline our current and potential approaches to this problem.

4 TOWARDS THE VERIFICATION OF HUMAN-AGENT TEAMWORK

4.1 Brahms → AgentSpeak → Model-Checker

On the one hand we have complex, human-agent teamwork, intended to be used in realistic environments. On the other, we have the need to formally verify teamwork, cooperation and coordination aspects of these human-agent teams. But these two seem far apart!

Our current approach can be described pictorially as in Figure 2. In essence it utilises the fact that the Brahms modelling language [17, 18] has been used for many years to model human-robot activities. Thus, Brahms modellers have significant experience in describing and modelling this type of system. Brahms describes teamwork at quite a high level of abstraction and characterises (simple forms of) human behaviour as agent behaviour.

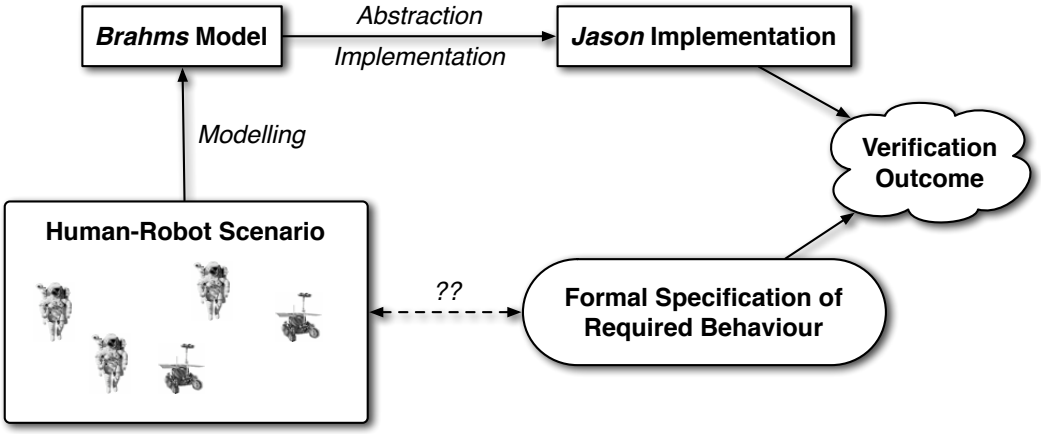

Figure 2: Our Current Approach.

The usual mode of use for Brahms is in modelling and simulating the activities of the system. However, verifying Brahms models directly is still much too difficult, due to the large and complex nature of Brahms models (see Section 4.3). Our approach here is to abstract from the large Brahms system and re-implement the required aspects in AgentSpeak (specifically, *Jason* [4]). This is possible since the (informal) semantics underlying Brahms is essentially the BDI theory; it is useful since it provides us with a smaller, executable description (in *Jason*) which is still relevant to the formal requirements we wish to check. Indeed, comparisons can be made between the Brahms simulations and the runs of the corresponding *Jason* code. Once we have *Jason* representing the required scenario, then we can analyse this with respect to the formal specifications, for example via model checking [3] as discussed above.

Unfortunately, establishing the formal correctness of the modelling and abstraction/implementation steps is not possible, due to the vagueness of Brahms semantics. However, the intention is that, if any error is found in the *Jason* code, when checking a certain property, then we attempt to re-generate a similar error within a simulation of the Brahms model. Such errors might be artifacts of the translation to *Jason*, but might also reflect real errors in the teamwork model.

4.2 Brahms → AIL → AJPF Model-Checking

There are currently *many* agent-oriented programming languages and agent platforms (often extensions of Java) [10]. Rather than providing an approach in which the complex logical properties of systems using just *one* particular agent approach can be verified, as in the work referenced above, we have developed a flexible framework allowing the verification of a wide range of agent-based programs, produced using various high-level agent programming languages. In recent work [1, 9], a Java library called AIL (Agent Infrastructure Layer) was developed which contains data structures and operations that generalise those used in the interpreters of various agent languages. Various optimisations have been done for when AIL-based agent language interpreters are verified using the Java Pathfinder (JPF) model checker for Java code (see more details about JPF below). This is what we referred to as the AJPF Model-Checker above.

Thus, another alternative to address the problem at hand is to re-implement Brahms using the AIL library, as this would allow us to take advantage of the AJPF optimisations. Even though AIL is very recent work, initial experiments show that using an AIL-based interpreter for an agent language immediately gives significant gains in terms of memory usage and speed of model checking. Of course, this approach would require the whole of Brahms to be re-implemented. Because Brahms has been developed for much longer than most other (modern) agent languages, it possibly involves a lot more code then our previous experiences, which showed that it is straightforward to re-implement an agent-language interpreter using AIL. However, this ease of implementation was to some extent due to the fact that the languages used so far had operational semantics, and the

semantic rules used in such formal semantics are directly mapped into AIL operations — as we discussed above, this is not the case with Brahms. On the other hand, Brahms is also Java based, so it might turn out to be possible to only re-implement the core classes of Brahms using AIL (and keep the other Brahms classes intact) and still take advantage of the AJPF optimisations. This and various other issues will have to be investigated in detail in our future work.

4.3 Brahms → JPF

JPF [22] is an explicit state model checker for runtime-based verification of Java bytecode (JPF is available *open source* at `http://javapathfinder.sf.net`). Essentially, JPF implements its own Java virtual machine on top of the host system's virtual machine — as JPF itself is written in Java, the "host virtual machine" is the Java virtual machine used to run JPF itself. The difference is that JPF virtual machine explores *all* possible paths that may be taken in a program's execution, continuously checking for deadlocks, violated assertions, and un-handled exceptions. To enable this, the state space of the program has to be finite, of course, and also within reasonable bounds, otherwise JPF runs out of memory (or the user runs out of time and patience before JPF runs out of memory). If JPF finds an error in one of the possible executions, it immediately reports to the user all the steps leading to that error — in model checking this is called the *counter-example* and is of great help for developers to fix bugs. Since it was released open source, JPF no longer supports checking of temporal logic formulae, which is the basis of the type of logic language we use to write specifications in agent verification; an important advantage of AIL is that is allows the use of linear temporal logic formulae in the properties to be checked by JPF.

Clearly, as Brahms is implemented in Java, another alternative is to "feed" the whole of the Brahms platform into JPF for verification of a particular model. Large agent platforms have a substantial amount of code that is irrelevant for the model checking process: parsing is a typical example. Even though JPF has internal abstraction mechanisms that can lead to impressive reduction of the state space, in our experience, the AIL optimisations, because they are specific to agent languages, lead to even greater reduction of the state space. So, realistically, using this approach means that all the optimisations already done in AIL would have to adapted and re-implemented for the Brahms code.

5 Concluding Remarks

In this paper we have described our approach to the formal verification of human-agent teamwork. Initial verification has been carried out following the approach described in Section 4.1, but we are not yet at the stage to fully verify large-scale Brahms models of astronaut-robot activity.

There are, of course, limitations to our approach. Most obviously, since the Brahms modelling language has no formal semantics, we must develop one. However, this semantics must be appropriate both to the type of scenarios considered and the form of model-checking to be used. Another possible limitation is that we do not consider low-level interactions/aspects, and so cannot directly verify control systems, etc (although it is questionable whether we wish to do this here). Although quite small verification tasks have been achieved, these are *not* fast. Thus, there will be a question of scalability to be tackled in the future.

One obvious aspect that we have not mentioned in our approach is that of *uncertainty*. Uncertainty and probability are key aspects that must (eventually) be modelled and checked. Agent models have been extended with various different probabilistic and fuzzy notions, though there is little application of these to human-agent teamwork. Similarly, while there are probabilistic model-checkers, such as PRISM [11], these are a long way from the agent models we would require. Another interesting avenue is to explore extending JPF with probabilistic aspects.

While the rational agent metaphor, comprising goals, beliefs and deliberation, is very flexible and intuitive, it is not sufficient to capture the core behaviours of multi-agent systems. Higher-level organisational aspects are often added to this model, for example roles, norms, team commitments, etc. Associated with this is the idea that, once an agent agrees to be part of a team it potentially loses some of its autonomy. This idea of adjustable autonomy is not yet present in the work we describe above.

Finally, we consider how our verification approach may be used in practice. Formal verification, particularly model-checking, is very successful if we have a comprehensive model of the system and its environment. However, for realistic applications, such models are typically both infinite and uncertain. So, typically, we abstract away from some aspects in order to get a smaller model, simplifying the problem but (hopefully) retaining key properties. Often we cannot guarantee to check every behaviour, as the abstraction taken might be too coarse. However, the formal modelling is also useful for generating tests for the infinite parts of the system. Thus, a combination of model-checking and testing, possibly also with theorem-proving for analysing infinite behaviour, is likely to be appropriate for such problems.

References

[1] R. H. Bordini, L. A. Dennis, B. Farwer, and M. Fisher. Automated Verification of Multi-Agent Programs. In *Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, September 2008

[2] R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Model Checking Rational Agents. *IEEE Intelligent Systems*, 19(5):46–52, September/October 2004.

[3] R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Verifying Multi-Agent Programs by Model Checking. *Journal of Autonomous Agents and Multi-Agent Systems*, 12(2):239–256, March 2006.

[4] R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-agent Systems in AgentSpeak Using Jason*. Wiley, 2007.

[5] W. J. Clancey. Simulating activities: Relating motives, deliberation, and attentive coordination. *Cognitive Systems Research* 3(3):471–499, 2002.

[6] W. J. Clancey, P. Sachs, M. Sierhuis, and R. van Hoof. Brahms: Simulating Practice for Work Systems Design. *International Journal on Human-Computer Studies* 49:831–865, 1998.

[7] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Dec. 1999.

[8] DART, 2006. `http://www.nasa.gov/mission_pages/dart/main`.

[9] L. A. Dennis, B. Farwer, R. H. Bordini, M. Fisher, and M. Wooldridge. A Common Semantic Basis for BDI Languages. In *Proc. Seventh International Workshop on Programming Multiagent Systems (ProMAS)*, volume 4908 of *Lecture Notes in Computer Science*, pages 124-139. Springer-Verlag, 2008.

[10] M. Fisher, R. H. Bordini, B. Hirsch and P. Torroni. Computational Logics and Agents: A Roadmap of Current Technologies and Future Trends. *Computational Intelligence* 23(1):61–91. Blackwell Publishing, Feb. 2007.

[11] A. Hinton, M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM: A Tool for Automatic Verification of Probabilistic Systems. In *Proc. TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 441–444. Springer, 2006.

[12] A. N. Leontev. *Activity, Consciousness, and Personality*. Prentice-Hall, 1978.

[13] A. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In *Proc. 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW)*, volume 1038 of *Lecture Notes in Computer Science*, pages 42–55. Springer, 1996.

[14] A. S. Rao and M. Georgeff. BDI Agents: From Theory to Practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS)*, pages 312–319, San Francisco, CA, June 1995.

[15] A. S. Rao and M. P. Georgeff. Modeling Agents within a BDI-Architecture. In *Proc. International Conference on Principles of Knowledge Representation and Reasoning (KR)*, Cambridge, Massachusetts, Apr. 1991. Morgan Kaufmann.

[16] Remote agent experiment, 1998. `http://ic.arc.nasa.gov/projects/remote-agent`.

[17] M. Sierhuis. *Modeling and Simulating Work Practice. BRAHMS: a multiagent modeling and simulation language for work system analysis and design*. PhD thesis, Social Science and Informatics (SWI), University of Amsterdam, SIKS Dissertation Series No. 2001-10, Amsterdam, The Netherlands, 2001.

[18] M. Sierhuis. Multiagent Modeling and Simulation in Human-Robot Mission Operations. (See `http://ic.arc.nasa.gov/ic/publications`), 2006.

[19] M. Sierhuis, J. M. Bradshaw, A. Acquisti, R. V. Hoof, R. Jeffers, and A. Uszok. Human-Agent Teamwork and Adjustable Autonomy in Practice. In *Proceedings of the 7th International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS)*, Nara, Japan, 2003.

[20] L. A. Suchman. *Plans and Situated Action: The Problem of Human Machine Communication*. Cambridge University Press, 1987.

[21] S. M. Veres and J. Luo. A Class of BDI Agent Architectures for Autonomous Control. In *Proceedings of the 43rd IEEE Conference on Decision and Control (CDC)*, volume 5, pages 4746–4751. IEEE Press, 2004.

[22] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering* 10(2):203-232, 2003.

[23] L. S. Vygotsky. *Mind in Society: The Development of Higher Psychological Processes*. Harvard University Press, 1978.

[24] M. Wooldridge. *An Introduction to Multiagent Systems*. John Wiley & Sons, 2002.

[25] M. Wooldridge and A. Rao, editors. *Foundations of Rational Agency*. Applied Logic Series. Kluwer Academic Publishers, Mar. 1999.