A Synergistic and Extensible Framework for Multi-Agent System Verification.

Josie Hunter
Oregon State University
Corvallis, Oregon, USA
josie@snowgoose.com

Franco Raimondi
MiddleSex University
London, UK
f.raimondi@mdx.ac.uk

Neha Rungta
NASA Ames Research Center
Moffett Field, CA, USA
neha.s.rungta@nasa.gov

Richard Stocker
University of Liverpool
Liverpool, Merseyside, UK
r.s.stocker@liverpool.ac.uk

ABSTRACT

Recently there has been a proliferation of tools and languages for modeling multi-agent systems (MAS). Verification tools, correspondingly, have been developed to check properties of these systems. Most MAS verification tools, however, have their own input language and often specialize in one verification technology, or only support checking a specific type of property.

In this work we present an extensible framework that leverages mainstream verification tools to successfully reason about various types of properties. We describe the verification of models specified in the Brahms agent modeling language to demonstrate the feasibility of our approach. We chose Brahms because it is used to model real instances of interactions between pilots, air-traffic controllers, and automated systems at NASA. Our framework takes as input a Brahms model along with a Java implementation of its semantics. We then use Java PathFinder to explore all possible behaviors of the model and, also, produce a generalized intermediate representation that encodes these behaviors. The intermediate representation is automatically transformed to the input language of mainstream model checkers, including PRISM, SPIN, and NuSMV allowing us to check different types of properties. We validate our approach on a model that contains key elements from the Air France Flight 447 accident.

Categories and Subject Descriptors

I.2.11 [**Distributed Artificial Intelligence**]: Multiagent systems; D.2.4 [**Software/Program Verification**]: Model Checking

General Terms

Verification, Multi-Agent Systems

DRAFT VERSION. To appear in: *Proceedings of the 12th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2013), Ito, Jonker, Gini, and Shehory (eds.), May, 6–10, 2013, Saint Paul, Minnesota, USA.*
PREPRINT

Keywords

Framework, Brahms, model checking, Java PathFinder

1. INTRODUCTION

In the past two decades multi-agent systems (MAS) have been employed successfully in modeling various applications, ranging from economic and financial markets to flight controllers. There is a growing demand for verification techniques that guarantee safety in a large number of these domains. Correspondingly, a number of tools and techniques have been developed to support agent-based development (e.g., using JADE [1]), agent-based simulation (e.g., using MASON [11] or Swarm [18]), and multi-agent systems verification (e.g. using the AIL/AJPF toolkit [4], or a dedicated model checker such as MCK [5], MCMAS [10], or Verics [9]). Verification of multi-agent systems is also performed by translating or encoding languages for multi-agent systems into traditional temporal-only model checkers such as SPIN [6] and NuSMV [2]. Current MAS verification techniques, however, do not include the advancements in the different mainstream verification tools and technologies within a single cohesive framework. Also, most MAS verification tools are designed for a specific input language and do not support analysis of different MAS formalisms. We believe that a framework that can support analysis of multiple MAS formalisms and can be integrated with existing verification tools easily can play a key role in further advancing MAS verification technology. To this end, we present an extensible framework that is not restricted to a single MAS input format and can leverage the capability of mainstream verification tools to enable the verification of different specifications and properties.

In this paper we present details of our framework using as an example the Brahms multi-agent simulation system and models [3, 14]. In the Brahms input language the various agents, objects, geography, facts and beliefs are modeled explicitly. Brahms also provides mechanisms to model interactions between agents and work processes. Brahms is currently used to model interactions between humans and automated systems in the context of aviation safety at NASA. The input to our framework is a Brahms model and an implementation of the Brahms semantics [15]. We then generate an intermediate representation that encodes all possible behaviors of the model by extending the Java PathFinder

(JPF) model checker [8]. The intermediate representation is automatically transformed into input formats to other verification tools such as PRISM, SPIN, and NuSMV. This automatic transformation allows us to reason about temporal properties and agent-specific modalities including the capability of reasoning about *probabilities* and *time bounds*.

To demonstrate the feasibility of our approach, we present a case study where we verify a Brahms model containing key conditions on board of the Air France AF447 flight that crashed in the equatorial Atlantic on June 1, 2009. One of the goals of the model is to provide validation to the claim in the accident report that the crash was due to pilot error and not to the weather conditions or hardware failures. Using the model we can successfully verify that even with incorrect airspeed readings from the Pitot tubes, a model of an experienced pilot can use other information available to recover from a stall and avoid the crash. We also compare our framework with a previous approach for Brahms verification described in [15] on a common example involving a domestic-health care scenario, showing that we can achieve a 5x speedup.

2. A MAS VERIFICATION FRAMEWORK

In this section we provide an overview of our framework, background on Brahms and JPF, and details of the MAS verification process within our framework. We provide descriptions of the key elements that allow our framework to be extensible and synergistic. We use a representative example to demonstrate the potential extensibility of our approach and discuss how we can leverage state of the art mainstream verification technologies efficiently.

2.1 Framework Overview

Fig. 1 presents a high level overview of our framework. The input to our framework is a MAS model and its corresponding semantics. In our work the input is a Brahms model along with an implementation of its execution semantics (Brahms simulator).

The *MAS connector* shown in Fig. 1 generates an intermediate representation of the MAS. The intermediate representation encodes all the relevant interactions between the agents in the MAS model. The MAS connector could be designed and implemented in several different ways. A probe that is injected into an execution environment such as MASON or Swarm is an example of a potential lightweight MAS connector. In our work, we implement the connector by extending the Java Pathfinder (JPF) model checker to control the execution of the Brahms simulator. The implementation, however, is designed to ensure that connectors within our JPF extension are *configurable* and *reusable* for other MAS systems. The use of extensible plug-ins in JPF such as creating customized choices for a given model allows the efficient generation of the interactions of interest for a given MAS model. Furthermore, the configurable nature of JPF provides end-users the ability to select the granularity of the interactions that should be encoded in the intermediate representation of the model.

The *intermediate representation* is an encoding of the different interactions between the agents in the MAS model; in our implementation, this is a graph that encodes all relevant interactions in a Brahms model as shown in Fig. 1. The nodes contain values for the facts and beliefs of the various agents in a Brahms model, while the edges represent conditions that lead to the update of the facts and beliefs. The MAS connector is used to gather and store additional information such as update probabilities, temporal and epistemic relations between different nodes, and labels for edges.

Different search and exploration strategies could be implemented in the MAS connector to further reduce the size of the behavior space of the MAS, e.g., abstraction techniques, symbolic representation using OBDDs, and customized state-matching algorithms. The intermediate representation is fully re-usable for different input MAS models and output formats. Furthermore, as the behavior space of the MAS model is generated, our framework allows on-the-fly verification of safety properties.

We have developed a set of automated translators to convert the intermediate representation into input formats for mainstream verification tools. We currently have translators for model checkers such as SPIN, NuSMV and PRISM, thereby enabling the verification of LTL/CTL properties, probabilities, time bounds and cost. We are currently developing a translator for MCMAS to reason about strategies.

2.2 Brahms

Brahms is a simulation and development environment originally designed to model the contextual situated activity behavior of groups of people in a real world context [3, 14]. Brahms has now evolved to model humans, robots, automated systems, agents, and interactions between humans and automated systems.

Brahms follows a rational agent approach: rational agents are autonomous, react to changes in circumstance and choose their actions based on their own agenda. Brahms has similarities to BDI (beliefs-desires-intentions) architectures [12, 13], which are used for modeling rational agents. BDI models describe the goals the agent has and the choices it makes, all based on a certain set of beliefs.

2.2.1 Brahms Models

A Brahms model contains a set of *Objects* and *Agents* that are used to model humans and automated system in a real-world context. Brahms is able to represent artifacts, data, and concepts in the form of classes and objects. A geography model is used to locate agents and objects and provides an additional context to activities.

The key aspects of Brahms are:

1. *attributes*: properties of agents/objects/locations,
2. *activities*: actions an agent performs, typically consume simulation time,
3. *beliefs*: each agent's own personal perception of all the *attributes* in the model,
4. *facts*: actual values of the *attributes*,
5. *detectables*: detection of *facts*, bringing facts into an agent's belief base and determining the response,
6. *workframes*: guarded plans which denote a sequence of events required to complete a task, including belief updates and actions,
7. *thoughtframes*: guarded plans for belief revisions made due to the situated context of the agent/object, and
8. *time*: forms the time-line output presented by Brahms to describe the simulation.

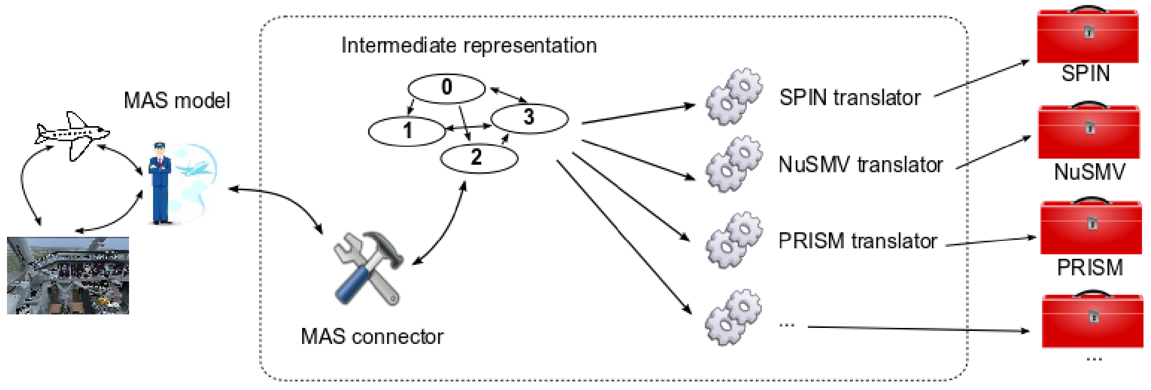

Figure 1: An extensible architecture that leverages state of the art technologies to verify MAS models.

2.2.2 Brahms Semantics

The flow chart in Fig. 2 presents an overview of the Brahms semantics and depicts how the agents advance in time and update the values of their facts and beliefs. Formal operational semantics for Brahms are detailed in [17, 16]. In this work, we use a Brahms Simulator implemented in Java that follows the high-level flow described in Fig. 2. The flow through the Brahms semantics starts with the box labeled $S1$ in Fig. 2. A Brahms scheduler here initializes the system, the agents, objects and geography. The scheduler instantiates all the agents as shown in $A1$, where they initialize their facts and beliefs. After initialization, the scheduler moves to $S2$ and instructs all agents to process their thoughtframes (box $A2$). In $A2$, the agents generate a set of active thoughtframes and then proceed to $A3$. In $A3$ the scheduler selects the active thoughtframe with the highest priority. If, however, there is more than one active thoughtframes with the same priority, an active thoughtframe is randomly selected from the set. The box is shaded to represent a possible *non-deterministic choice*. The process is repeated until there are no more active thoughtframes after which the scheduler moves to $S3$. After executing $S3$ the scheduler instructs the agents to move to $A4$ to generate their set of active workframes. Agents then proceed to $A5$ to select a workframe.

Workframes contain activities (represented by $A7$) and conclude statements ($A8$). If the event is an activity, the scheduler randomly selects a duration of the activity between the minimum and maximum durations specified for the activity, then inform $S4$ of this duration. If the event is a conclude then the value of a belief may be updated depending on the belief condition; the belief condition is a percentage representing the chance the belief update will occur. The agent cycles between $A6$ and $A8$ until it finds an activity ($A7$) or until the workframe finishes. The scheduler waits in $S4$ until all activity durations have been received, if an agent has no active workframes then a duration of 0 is sent (from $A4$ to $S4$). The scheduler then calculates the shortest duration in $S4$ and moves the clock forward by this amount. If the duration is 0 (i.e. all agents have no active workframes) then the system terminates in $S6$. If the duration is not 0, then agents are informed to deduct the value from their activity duration in $A9$. The cycle then returns to $S2$ and continues until all agents have no active workframes.

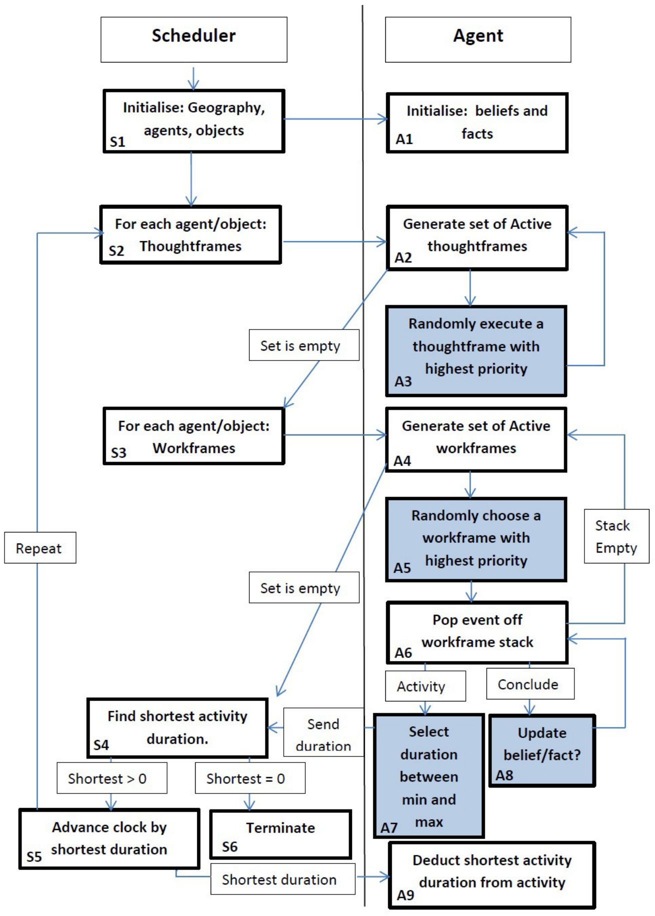
Scheduler
Agent
S1 Initialise: Geography, agents, objects
A1 Initialise: beliefs and facts
S2 For each agent/object: Thoughtframes
A2 Generate set of Active thoughtframes
A3 Randomly execute a thoughtframe with highest priority
Set is empty
S3 For each agent/object: Workframes
A4 Generate set of Active workframes
A5 Randomly choose a workframe with highest priority
Stack Empty
Repeat
Set is empty
A6 Pop event off workframe stack
Activity
Conclude
S4 Find shortest activity duration.
Send duration
A7 Select duration between min and max
A8 Update belief/fact?
Shortest > 0
Shortest = 0
S5 Advance clock by shortest duration
S6 Terminate
Shortest duration
A9 Deduct shortest activity duration from activity

Figure 2: Overview of Brahms Semantics

The simulator can be used as a stand-alone tool to perform simulation. In our case, we employ it in conjunction with our MAS connector based on Java Pathfinder.

2.3 Java PathFinder: a Configurable JVM

We have implemented our MAS connector for Brahms using Java Pathfinder. Java Pathfinder (JPF) [8] is an extensible Java Virtual Machine (JVM) that enables analysis, execution, and verification of Java bytecode. JPF was originally developed as a concrete-value, explicit-state software model checker for concurrent programs. In the past few years, however, JPF has evolved into a extensible Java analysis framework for developing and exploring different verification techniques and application domains. Furthermore, there are several JPF extensions based on symbolic execution, BDD-based analysis, parallel search, and abstraction-based techniques. Some of these extensions are fairly mature prototypes that have been used to make verification of Java program tractable and have the potential to be used in MAS verification as well for better scalability.

The inputs to JPF are a set of class files for the system being analyzed, a set of configuration files that specify the desired JPF execution mode, program properties to verify, and artifacts to generate. JPF supports search with backtracking, state matching, and non-determinism in both data and scheduling decisions. JPF constructs the program state space on-the-fly during the execution of the program in a customized virtual machine. In this work, the input to JPF are the class files for the Brahms simulator and the Brahms model and the output is graph for the intermediate representation.

JPF uses a generic model of the program state space consisting of *States*, *Choices* and *Transitions*. *States* are restorable snapshots of a program execution along a particular path. The snapshot contains the heap of the program, its current program location, and the current operand stack. *Choices* are the mechanism to differentiate between different possible executions leading out of a state, for example, different possible data values that can be assigned to input variable by a user or by a random variable. A *transition* is a sequence of bytecode instructions where the first and last instruction in the sequence represents a non-deterministic choice. At every transition boundary, JPF saves the current JVM state (the program state) in a serialized form for the purpose of backtracking and state matching.

2.4 Efficient & Extensible Choice Generation

The MAS controller in our framework systematically explores all possible (non-deterministic) choices that occur in the input MAS model. We leverage the choice generation mechanism within JPF to interact with the Brahms simulator and generate all possible choices. The ability of JPF to configure the granularity of choices is a key element for our ability to deal with real-life scenarios. Indeed, most model checkers do not provide user-configurable options to manage different types of non-determinism.

This section describes how different types of choices are created and explored in JPF. When JPF analyses a Brahms model along with its semantics, a new state is only generated when the search within JPF encounters a point of non-determinism. The three main points of non-determinism in the Brahms semantics shown by shaded boxes in Fig. 2 are:
1) $A8$: Fact and Belief updates
2) $A7$: Activity Durations
3) $A3$, $A5$: Workframe and ThoughtFrame choices

A new JPF state may be generated at $A3$; however, before another state is generated at $A7$, millions of bytecode instructions could be executed. Customized choice generators in JPF systematically explore all the possible choices at only points of non-determinism relevant to Brahms semantics, and turn off all other points of non-determinism that may exist in a general Java program. This allows us to generate a behavior space that is customized, based on the requirements of the MAS being analyzed.

2.4.1 Fact and Belief Updates

The updates to facts and beliefs of agents and objects are made using *conclude* statements in a Brahms model. An example of a conclude statement is:
$conclude((Pilot.checkStall = false), bc : 70, fc : 70)$;
This states that the belief and fact `checkStall` in the `Pilot` agent will be updated to *false* with a probability of 70%. Here `bc` represents belief certainty while `fc` represents fact certainty. Intuitively, it is the certainty with which the updates to the beliefs and facts occur. When the simulation in Brahms encounters a conclude statement (the parent of $A8$ in Fig. 2), it checks next whether the belief and fact should be updated. The simulation in Brahms flips a weighted coin to decide whether the fact and belief are updated or not. The code in the simulator implementation for the $A8$ in Fig. 2 is as follows:

```
Random rgen = new Random();
public boolean update (int certainty) {
 if(certainty == 100) return true;
 if(certainty == 0) return false;

 int random = rgen.nextInt(98)+1;
 if(certainty >= random)
     return true;
 return false;
}
```

The update method gets as input a belief certainty or a fact certainty as input. When the certainty is less than 100 and greater than 0, a random number in the range 1 and 99 is generated. If the certainty is greater than equal to the random number the method returns true, otherwise it returns false. There is support for generating choices in JPF where a random number is generated and used in a Java program. Note that when the certainty is 0 or 100 the method returns and no choices nor states are generated by JPF; however, for all other values of certainty, a data choice generator in JPF creates 99 choices. For each one of the 99 choices, a new JPF state is created where the variable, `random`, is respectively assigned a value in the range $[1 \ldots 99]$. Based on the value assigned to `random` the method returns either true or false. As a result, we know that for a `conclude` construct in Brahms there are only two main branches of interest. One, when an update is made to belief and fact in the conclude statement; two, when there is no update to the belief and fact in conclude statement.

We configure JPF to use a custom choice generator *in lieu* of its standard data choice generator to efficiently explore choices with respect to the MAS analysis. We have implemented a new custom choice generator that (a) creates a single choice when the certainty is 100 and the method returns true (b) creates a single choice when the certainty is 0 and the method returns false (c) creates two choices for certainty values in the range 1 and 99 where the method returns true and false respectively. In our MAS connector we monitor the execution of different methods through the *observer* patterns implemented in JPF as listeners. In our framework we intercept the invocation of the `update` method, skip its execution, and assign its return value based on the custom choice choices.

2.4.2 Activity Duration

The activities in Brahms have a duration in seconds associated with them. The duration of the activity can either be fixed or can vary based on certain attributes of the activities. When the *random* attribute of an activity is set to true the simulator randomly selects the activity duration between the min and max durations specified for the activity. In order to completely explore all the possible choices we create a choice generator that can systematically explore all possible durations for the activities between the min and max durations. In certain cases this may cause the state space to explode. To mitigate the state space explosion, various heuristics are available to the users where they can select the number of choices that are to be explored, for example, one heuristic explores the min, max and median durations for the activity.

2.4.3 WorkFrame and ThoughtFrame choices

The third and final point of non-determinism in Brahms arises when a set of active workframes or a set of active thoughtframes have the same priorities. As described in Section 2.2, the workframe with the highest priority is executed from the set of active workframes as shown in $A5$ of Fig. 2. However, there may be several workframes that have the same priority. In this case a workframe is randomly selected to be executed (the same situation could occur for thoughtframes with equal priority).

In our framework we create a custom choice generator for systematically exploring all workframe and thoughtframes with the same priority. When the method to select the workframe and thoughtframe is executed in the Brahms simulator, the listener in our MAS controller intercepts the call to create choices. The number of active workframes with the same priority is the number of choices created by JPF. Again, the same mechanism is implemented for active thoughtframes with the same priority.

2.5 Intermediate representation

A key element of our framework is the notion of generalized *intermediate representation*, rather than a custom language for MAS. The intermediate representation is a simple reachability graph $G := \langle N, E \rangle$ where N is the set of nodes and E is the set of edges. This graph is generated by the MAS connector. Each node $n \in N$ is labeled with the belief/facts values of the agents and objects. An edge between the nodes represents the updates to beliefs/facts and is also labelled with probabilities. The *reachable states* generated by the JPF MAS connector are mapped to the nodes in the intermediate representation.

The nodes in the intermediate representation only contains the part of the JPF state relevant for verifying properties about the Brahms model. Recall that the state generated in JPF contains an entire snapshot of the program execution including the heap and the operand stack. In essence JPF state contains all the elements of a Brahms model, since the input to JPF is the implementation of the Brahms semantics (we will refer to it as the Brahms simulator for brevity); the data structures part of the Brahms simulator are also part of the JPF state.

We create and store a node in the intermediate representation for a JPF state that is generated that only contains the values of the beliefs and facts of agents and objects. Several JPF states may map to a single abstract state. Consider the sequence of JPF states generated along a path $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4$. The $\rightarrow$ represents transitions between states. Suppose, states s_0, s_1 and s_2 map to the node n_0 while s_3 and s_4 map to n_1 then the corresponding edge in the intermediate representation is $n_0 \rightarrow n_1$. Note, however, that the state matching in MAS connector is performed on the JPF states in order to explore all possible behaviors of the MAS model.

We also store other details about transitions in JPF such as the transition probabilities and action labels on the edges of the intermediate representation. Consider the *conclude* statement reported earlier where `Pilot.checkStall` is set to *false* with belief certainty 70%. The choice generation creates two choices, one where the belief is updated (with probability 70%), and one where it is not update (with probability 30%). One edge in the intermediate representation is annotated with the probability of 0.70: $n_i \rightarrow^{.70} n_j$; while another edge in the intermediate representation is annotated with the probability of 0.30%: $n_i \rightarrow^{.30} n_i$. If three choices are generated for three active workframes with the same priority, each transition probability is 1/3.

2.6 Verification

The verification of safety properties and other reachability is performed on-the-fly as new states and transitions are generated in JPF. Additional verification activities can be performed on the intermediate representation after all the JPF states have been generated.

We have developed a set of *translators* to translate automatically our intermediate representation to an input problem for a range of model checkers. For instance, a user can select to output a PRISM model to verify (or compute) probabilities of certain events, or time bounds for certain events to happen. The intermediate representation is a simple reachability graph with the belief/facts values as labels and edges representing updates to beliefs/facts. The corresponding PRISM models do not require knowledge of the Brahms Java encoding and the connector builds the labels automatically. The probabilities on the edges in the intermediate representation is also added to the PRISM model. Similarly, the SPIN translator can generate PROMELA code to be used in conjunction with SPIN to perform LTL verification, or NuSMV code to enable BDD-based CTL verification. In all cases, our translators only extract the values that are relevant for the target verification tool.

The goal of this work is not to generate the optimal encoding of intermediate representation for a model checker but rather be able to verify different types of properties. An interesting avenue of future work would be to study how the current intermediate representation could be improved and tailored to specific model checkers.

3. EXPERIMENTAL RESULTS

In this section we present empirical results to demonstrate the utility of our MAS verification framework in analyzing and verifying Brahms models. First we present a case study using the scenario from the Air France 447 crash. This provides validation of the fact that the framework such as the one described in this paper can be used for the verification of models that describe interactions relevant in aviation safety. We then perform a direct quantitative comparison with another approach [15] that translates Brahms into Promela models directly, by encoding both the model and the semantics as a Promela model. This approach verifies properties using the SPIN model checker.

3.1 Case study: Air France 447 Model

On June 1, 2009 the Air France Flight 447 between Rio de Janeiro and Paris crashed in the equatorial Atlantic. The final BEA accident report[1] states that the weather conditions caused icing to build up on the Pitot tubes resulting in inaccurate airspeed readings. The inexperience of the pilot was determined to be the cause of the crash. The pilot in charge misjudged the airspeed of the plane and increased the altitude of the plane without realizing the plane was in a stall which eventually led to its crash. According to the report the pilot was presented with several chances to recover, but, was unable to do so.

A model of the conditions during the flight of AirFrance 447 was developed to validate that the conditions or hardware failures did not lead to the crash[2]. We refer to this as the "flight model". This flight model only includes the important factors (i.e. those that have been determined as catalysts in the cause of the crash). In the model, there are several components that interact with one another, namely: the pilot, the controls, the airplane itself, two Pitot tube sensors, the weather, and the stall level. The pilot uses the controls (throttle, elevators) to manipulate the speed, altitude and attitude (i.e., the inclination with respect to the wind) of the plane. In the model, the pilot relies on the airspeed reading, which is provided by the Pitot sensors. Pitot sensors monitor the speed of the plane and relay that information back to the pilot through the gauges. The `Weather` object in the model can simulate stormy conditions which results in ice to form over the Pitot tubes. In our model of the Pitot tubes, when the amount icing over the sensors goes above a certain threshold, the airspeed readings become inaccurate. If the pilot notices that the airspeed readings from each of the two Pitot sensors do not match, the pilot attempts to estimate the airspeed using measures described in aviation procedures. In the model the pilot has access to the information that could allow him to determine that the airspeed values reported by the Pitot sensors are incorrect and has options to estimate the current airspeed. The stall level may increase depending on the combination of airspeed, altitude, and attitude. Once the stall level goes above a certain threshold, through different mechanisms the pilot becomes aware of the situation and can adjust the controls accordingly.

We encode this model as a Brahms model. We implement all the different agents and objects and their interactions through activities, thoughtframes, workframes, and updates of beliefs and facts. As an example, part of a workframe from the Pilot agent in the Brahms model is shown below:

```
workframe wf_stallAdjust {
repeat: true; priority: 2;
when(
knownval(current.time < 200) and
knownval(Stall.stallLevel > 0.5))
do {
adjustForStall();
conclude((Controls.elevators =
                 Controls.elevators - .04),
                 bc: 100, fc: 100);
           } }
```

The first variable (repeat) tells the simulator whether or not this workframe is repeatable, and the second gives the level of priority (in case more than one workframe is available). The two knownval statements within the when() clause are guards. The first guard, current.time < 200, simply provides a bound on the length of the simulation. The second guard checks if the stallLevel is greater than the threshold, 0.5. If both guards evaluate to true, then the events in the do clause are executed. The first event is a primitive activity named "adjustForStall", and the second event sets the elevators to decrease by .04. The second event updates the pilot's belief about the value of elevators with 100% probability (bc:100), and with 100% probability results in an actual change of the fact `Controls.elevator` (fc:100).

We can successfully validate the model to show that the pilot can always correct the stall in a timely manner and that the plane does not crash due to hardware failures. We verify this property using on-the-fly verification in JPF as the intermediate representation is being generated. It takes 2.5 minutes to generate the $28,648$ reachable states required to verify the property using JPF. Notice that this is a complex example involving several agents and objects, most of which have variables ranging over a continuous domain (e.g. altitude, airspeed, etc.). We are currently working in conjunction with aviation safety experts to create different versions of the model to increase the variability of the actions (especially for the pilot). Our aim is to study what conditions can lead to a crash, and estimate the probability of those conditions as well.

3.2 Comparison with another approach

Stocker et al. present a technique to translate a Brahms model directly into a Promela model and verify the Promela model using the SPIN model checker [15]. They use the formal semantics described in [17] as a basis for this translation. Note that we use exactly the same semantics in our Brahms simulator. First, they produce a Java representation of the Brahms model. The Java representation is a syntactic transformation of the Brahms model and the Brahms elements such as agents, objects, workframes, thoughtframes, beliefs, and facts into Java data structures such as lists and maps. Then, using this Java representation they produce Promela process descriptions, which represent partial instantiations of the semantics, suitable for input to the SPIN model checker. LTL specifications for the Brahms model are then be checked using SPIN.

[1]`http://www.bea.aero/en/enquetes/flight.af.447/flight.af.447.php`
[2]Statistical Analysis of Flight Procedures, by Adrian Agogino and Guillaume Brat at NASA Ames Research Center, CA.

The direct translation of Brahms to Promela can pose some challenges because Promela does not provide a natural way to encode object-oriented hierarchical systems such as the ones employed in Brahms. Furthermore, Promela does not support object orientation, stacks, strings, method calls etc., which only allows a partial instantiation of the Brahms operational semantics. This makes it difficult to prove that the translation matches the operational semantics. The SPIN model checker then generates choices at all the guards in order to generate the model. The lack of extensible plug-ins within SPIN makes it hard to create customized choices. In our framework we avoid this issue by encoding the reachable state space of the model using our intermediate representation, which is then translated into Promela.

3.2.1 Comparing results

The work in [15] uses a domestic-health care example to validate the approach. The model includes an elderly person, a helper robot, a care provider who is human, and another automated agent. The automated agent and robot assist the elderly person by issuing reminders and assisting them in their activities. When the automated agent and robot are unable to assist the person, they call the human care provider.

The work in [15] verifies ten different properties for this model. We analyze the same domestic-health care example using the same input Brahms model in our framework in order to empirically compare the effectiveness of our approach with the one presented in [15].

The MAS connector in our framework takes 54 seconds to explore a total of 7792 JPF states to generate the intermediate representation of the domestic-health care example in [15]. In the resulting intermediate representation there are 511 nodes with 690 edges. During the analysis of the domestic-health care model in JPF over 77 million bytecode instructions are executed, while 168 classes and over 2000 methods are analyzed. This demonstrates that the program analyzed is of a significant size. However, a mere 7792 JPF states are generated and only 511 nodes in the intermediate representation are produced as output.

The properties specified in [15] can be verified using SPIN after automatically transforming the intermediate representation into a Promela model. All the properties can be verified in a total of two seconds. Note that the SPIN verification is extremely fast because it does not need to allocate space for variables that may not be reachable. As reported in [15], the full model generation in SPIN takes longer due to the structure of the example and semantics. We report below the *total* time taken for the verification of this example using the two approaches. This time includes all the steps: in our cases these comprise intermediate representation generation using JPF, translation to SPIN, and SPIN execution time (for all properties). In the case of [15] the total time includes Java data structure generation from the Brahms source code, Promela code generation, and SPIN execution time (for all properties):

- Total time by [15]: 5 mins.
- Total time by our framework: 1 min

All the experiments were executed on the same machine (a standard quad-core Linux machine with 8 GB of RAM); in our case, the dominant component is the reachable state space generation using JPF (54 seconds); in the case of [15] the dominant component is the SPIN execution time.

As an additional validation step we have also encoded the properties of [15] both as CTL and PCTL properties, and translated our intermediate representation to NuSMV and PRISM code. As above, the final translation and verification steps take around 2 seconds at most for both model checkers.

4. RELATED WORK

There are several model checking tools available for multi-agent system verification. MCK, [5], allows verification of temporal-epistemic logics using various OBDD- and SAT-based techniques. MCMAS, [10], is an OBDD-based model checker that supports the verification of time, knowledge, and strategies. VERICS, [9], allows the verification of MAS specified with a rich input language based on Petri Nets, and uses bounded model checking among other techniques for verification. All these tools have a dedicated input language and, to the best of our knowledge, do not offer cross integration with different formalisms.

The AIL/AJPF toolkit, [4], is a closely related work to the verification framework presented in this paper. This toolkit employs JPF to perform verification of BDI agent programming languages. AJPF translates agent programming languages to an intermediate representation called the Agent Infrastructure Layer (AIL). This intermediate representation is then executed in AJPF, an extension of JPF. The key differences with our approach are: (1) We do not translate input into an intermediate representation. Instead, we operate directly on model in conjunction with the simulator for the model (2) We generate an intermediate representation for the *output* of our tool: this allows us to generate automatically input files for a range of state-of-the art model checkers. (3) Our MAS framework enables the verification of various languages (LTL, CTL, PCTL, ATL, etc.), while AIL/AJPF is limited to a BDI extension of LTL. (4) In our work JPF is used primarily for its backtrackable search, customizable choice generators, and support for customized state storage technique and not just as a verification tool. We could replace JPF with another MAS connector to generate the behavior space of the MAS model.

Another closely related work, [7], interacts with the interpreter for the GOAL MAS programming language to perform model checking. Our approach is different in that our framework supports multiple input formalisms, and we do not implement model checking algorithms directly; instead, we employ state-of-the-art tools to support multiple modalities.

Finally, as described in Section 3, the work by Stocker et al. [15] deals with the verification of Brahms models using a translation to SPIN. Our work is different in that we provide a more general framework that allows multiple input formalism and allows more expressive specification patterns (for instance using probabilities in PRISM). Additionally, as shown by the experimental results, we provide a more efficient mechanism for Brahms verification: the state space exploration performed by our framework only considers the components of the simulation that are relevant for verification (i.e. facts and beliefs), thus producing a more compact representation that is then fed into SPIN.

5. CONCLUSION

We presented a framework for the verification of multi-agent systems that allows us to leverage state-of-the-art verification tools. Our framework makes use of *connectors* to interact with existing multi-agent systems, generating an intermediate representation (behavior space). The behavior space can then be translated to the input language of mature verification tools such as SPIN and NuSMV. In contrast with other previous approaches we do not translate the *input* models into a common formalism. Instead, we provide an extensible framework that can support different inputs and can be integrated with existing tools. We validate the effectiveness of our framework using the Brahms agent modeling language, using JPF as a connector, and providing automated translators to SPIN, PRISM, and NuSMV. The experimental results from two case studies demonstrate the potential of the framework to tackle large state spaces.

In our current MAS connector implementation we store states explicitly: an avenue of future work is to explore the use of symbolic execution and abstraction-based techniques for further improving the scalability of the framework. As mentioned in the paper, JPF has several extensions for symbolic execution, parallel search, abstraction based analyses among others: we plan to integrate these extensions within our framework in our future work. Currently we are working on other automated translators to inputs for model checkers that reason about strategies (e.g. MCMAS). We are also working at the development of connectors for other formalisms, in addition to Brahms. Given the flexibility of our approach and the set of libraries already developed, we hope that the MAS verification community will adopt our framework and contribute toward the continued development of an extensible framework for MAS verification.

6. REFERENCES

[1] F. L. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE.* Wiley, 2007.

[2] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV2: An open-source tool for symbolic model checking. In *CAV*, volume 2404 of *LNCS*, pages 359–364. Springer-Verlag, 2002.

[3] W.J. Clancey, P. Sachs, M. Sierhuis, and R. Van Hoof. Brahms: Simulating practice for work systems design. *International Journal of Human-Computer Studies*, 49(6):831–865, 1998.

[4] L. A. Dennis, M. Fisher, M. P. Webster, and R. H. Bordini. Model checking agent programming languages. *Autom. Softw. Eng.*, 19(1):5–63, 2012.

[5] P. Gammie and R. V. D. Meyden. MCK: Model checking the logic of knowledge. In *Proceedings of CAV-2004, Lecture Notes in Computer Science*, pages 479–483. Springer, 2004.

[6] G. Holzmann. *Spin model checker, the: primer and reference manual.* Addison-Wesley Professional, first edition, 2003.

[7] S. S. Jongmans, K. Hindriks, and M. V. Riemsdijk. Model checking agent programs by using the program interpreter. *Computational Logic in Multi-Agent Systems*, pages 219–237, 2010.

[8] Java PathFinder. `http://babelfish.arc.nasa.gov/trac/jpf/`. Accessed: 15 October 2012.

[9] M. Knapik, A. Niewiadomski, W. Penczek, A. Pólrola, M. Szreter, and A. Zbrzezny. Parametric model checking with verICS. In *Transactions on Petri nets and other models of concurrency IV*, pages 98–120. Springer-Verlag, Berlin, Heidelberg, 2010.

[10] A. Lomuscio, H. Qu, and F. Raimondi. MCMAS: A model checker for the verification of multi-agent systems. In *CAV 2009*, Lecture Notes in Computer Science, pages 682–688. Springer, 2009.

[11] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan. MASON: A multiagent simulation environment. *Simulation*, 81(7):517–527, July 2005.

[12] A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI-architecture. In James F. Allen, Richard Fikes, and Erik Sandewall, editors, *KR*, pages 473–484. Morgan Kaufmann, 1991.

[13] A. S. Rao and M. P. Georgeff. BDI agents: From theory to practice. In Victor R. Lesser and Les Gasser, editors, *ICMAS*, pages 312–319. The MIT Press, 1995.

[14] 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.

[15] R. Stocker, L. Dennis, C. Dixon, and M. Fisher. Verification of brahms human-robot teamwork models. In *Proceedings of 13th European Conference on Logics in Artificial Intelligence*, JELIA'12, 2012.

[16] R. Stocker, M. Fisher, L. Dennis, and C. Dixon. A Formal Semantics for the Brahms Language. (See `http://www.csc.liv.ac.uk/ rss/publications`), 2011.

[17] R. Stocker, M. Sierhuis, L. Dennis, C. Dixon, and M. Fisher. A formal semantics for brahms. In *Proceedings of the 12th international conference on Computational logic in multi-agent systems*, CLIMA'11, pages 259–274, Berlin, Heidelberg, 2011. Springer-Verlag.

[18] Swarm Software for Agent-based Modeling. `http://www.swarm.org/index.php/Main_Page`. Accessed: 15 August 2012.