Formal Verification of ALICA Multi-agent Plans Using Model Checking

Thanh Binh Huynh
Distributed Systems Group
University of Kassel
Kassel, Germany
vtn@vs.uni-kassel.de

Nugroho Fredivianus
Distributed Systems Group
University of Kassel
Kassel, Germany
nfr@vs.uni-kassel.de

Huu Tam Tran
Distributed Systems Group
University of Kassel
Kassel, Germany
tran@vs.uni-kassel.de

Kurt Geihs
Distributed Systems Group
University of Kassel
Kassel, Germany
geihs@vs.uni-kassel.de

Thi Thanh Binh Huynh
Hanoi University of Science and Technology
Hanoi, Vietnam
binhht@soict.hust.edu.vn

ABSTRACT
In multi-agent systems (MAS), plans consisting of sequences of actions are used to accomplish the team task. A critical issue for this approach is avoiding problems such as deadlocks and safety violations. Our recent work addresses that matter by verifying plans composed in a language called ALICA (A Language For Interactive Cooperative Agents) that controls the agents' behavior. The investigation is conducted by creating a translation tool that implements an algorithm for translating ALICA plans into the format used by the real-time model checker UPPAAL. We tested our concept using several cases, and the result is promising to get further insight on multi-agent model checking.

CCS CONCEPTS
• General and reference → Verification;

KEYWORDS
 Verification, ALICA, model checking, multi-agent planning.

ACM Reference Format:

1 INTRODUCTION
ALICA (A Language For Interactive Cooperative Agents) is a behavior specification language for teams of agents. Since 2009, it has been employed on our soccer robots and demonstrated its capabilities, e.g., our robot soccer team achieved the third place in the RoboCup MSL at the European Open in 2016. ALICA is also proven to be useful for various application domains, e.g., Robotic Space Missions and Autonomous Driving [17]. However, even the best laid plan must be verified [15], and this is also true for ALICA plans. Our aim is to ensure that no unexpected events will arise. Using formal methods, each plan will be checked to meet given properties and to minimize the likelihood of any dangerous or undesirable effects, particularly, when we apply agent systems to safety-critical applications such as autonomous vehicles or spacecraft control.

A popular verification technique is model checking[5]. Model checking is a technique originally developed for verifying concurrent systems. Although model checking has been widely used to verify hardware systems, it is also extensively applied to verify software systems, as well as protocols [2, 8].

In this work, we present a set of results obtained from a model checking technique for the verification of plans written in ALICA. The task is quite challenging due to a number of constraints. Firstly, the verifier must convert ALICA plans to a model that describes the plans in sufficient detail. And secondly, the model should be sufficiently simple to be verified by the model checker. Moreover, several states in ALICA plans may contain more than one behavior for multitasking purposes, making it more difficult to be examined. To solve this matter, we compose a translation algorithm and create a corresponding tool with the ability of performing a proper conversion process.

For that task, the model checker UPPAAL is chosen due to its success in several verification cases for real-time systems [1, 10]. Therefore, we demonstrate our tool by presenting two main contributions. Firstly, we have constructed an algorithm called ALICA2UPPAAL for translating ALICA plans into the syntax used in UPPAAL. Secondly, we present our translation tool called A2U along with the technical description.

The rest of the paper is organized as follows: in Section 2, related works are presented. Section 3 briefly describes the general structure of an ALICA plan and the UPPAAL model checker. Then, we introduce our algorithm and the A2U tool in Section 4, followed...
by the discussion regarding the experimental results from our implementation in Section 5. Finally, Section 6 concludes the paper and points to the future work.

2 RELATED WORK

Various model checking techniques have been used in MAS to verify plans based on several formalisms. The work in [9] presents UPPAAL for HSTS, a planner and scheduler of the remote agent autonomous control system that is deployed in deep space. Then, Xu et al. [18] introduce an approach for modeling and verifying multi-agent behaviors using Predicate/Transition Nets. In [6], a method is proposed to verify non-deterministic execution plans for MAS using Hierarchical Colored Petri Nets, based on guidelines for modeling and verification activities. Barringer et al. [3] propose an approach applied in a NASA mission to verify command sequences created on the ground, against a set of flight rules, before sending them to a satellite. Such a command sequence has the same characteristics as a plan: it is a sequence of actions (commands) to be executed on board the satellite. Also, they have to satisfy the properties used in the flight rules, formulated as monitors in the TraceContract tool.

Vandi Verma et al. [16] present The Plan Execution Interchange Language (PLEXIL), a language for describing the behavior of an autonomous agent. It is used in the area of robotics, unmanned vehicles, automation in industrial operations, and software agents. PLEXIL also provides the combination with many high-level planners and supports the integration of domain descriptions as well as high-level decision making. To some extent, the expressiveness of the language is similar to ALICA. PLEXIL can express plans with loops, conditions and further control structures. Moreover, the formal semantics of PLEXIL enables a validation of plans and provides a framework for formal analysis [7]. However, PLEXIL was designed to express the behavior of a single agent system, which is different from ALICA that supports multi-agent concepts like synchronizations and task cardinalities. Hence, the verification process for ALICA plans is more difficult and complicated.

An earlier attempt to verify ALICA multi-agent plans was conducted in [11], where Answer Set Programming (ASP) is used to verify a number of plans during the modeling process. That work can help the designers who are unfamiliar with the concepts of multi-agent plans in avoiding errors when modeling an ALICA program. However, since the verification does not consider the correctness properties, the chance of creating unsafe plans still exists. The approach proposed in this paper can exhaustively explore a set of reachable states (i.e., the state space). Therefore, the designers are able to know that the properties may or may not hold for any created plans, before applying them in real agents.

3 BACKGROUND

So far, ALICA has been used as a language to implement various scenarios in a number of our works, e.g., soccer robots, exploration agents, and other mobile entities. The result is satisfying, indicated by the functioning system. However, some erroneous occurrences still appear now and then. For instance, a robot that should perform a free kick in the soccer game keeps waiting for any visible teammate until the time expires. Such a situation may appear when the opponent team implements a strategy to cover the teammate’s visibility. This is an example where the necessity of having the plans checked before being implemented to the robots arises, and UPPAAL is chosen for checking the plans beforehand, due to its versatility of performing the required examinations, as described in the following subsections.

3.1 ALICA Plan

Each ALICA plan consists of finite state machines, the lowest level of the structure that describes sequences of actions performed by the agents. A plan has one initial state and a number of regular ones. Every state can contain a set of behaviors and states are connected to each other through transitions. An agent that occupies such a state obligates to execute all behaviors in this state. ALICA considers a behavior as an atomic activity. For instance, in the area of mobile robots, typical behaviors are Drive and Wait. More details regarding ALICA can be found in [12–14].

Figure 1a illustrates a scenario where a team of mobile robots is assigned to collect samples of objects (e.g., rocks) by picking and bringing them to the base. Each agent is given the freedom of performing these activities:

1. **Drive**: going out of the base and search for an object
2. **PickUp**: attempting to pick up an object, in case of failure, it will attempt to redo the action
3. **CheckStatus**: checking the feedback from a sensor that informs the agent, whether the pickup process was successful or not.

We created an ALICA plan shown in Figure 1b for each robot to perform those activities. In this case, the typical behaviors are shown using instances like **Drive** and **Pickup**. A transition that connects a state to the other can "fire" if, and only if, all the corresponding conditions are satisfied. As illustrated in the figure, an agent (represented by a red circle) inhabits the state **MetObj** while executing a behavior to pick up the objects, before entering the state **GetObj**. In ALICA, a plantype (denoted by a purple icon) contains a number of plans that should be executed by the agent. For this example, we use the plantype **CheckStatus** for simulating a sensor and signal transmission between it and the agent. The plantypes can be inserted into states of the finite state machines. ALICA plans may contain several plantypes without any limitation. However, in Figure 1b, the **CheckStatus** is a given plantype that contains only one plan (see Figure 2).

An ALICA plan typically contains more than one finite state machine, where each of them is marked with a task and is limited by a minimum and maximum cardinality. The minimum cardinality is the minimum number of agents that are required for executing the plans and the maximum cardinality is the maximum number of agents, which are allowed to execute the plans. To provide a clear example, in robotic soccer only one agent is allowed to do a free kick, making the maximum cardinality equal to one. In other situations like defending an open play, we can have all agents perform as a defender. In the example illustrated here, the minimum and maximum cardinalities are equal to one. Hence, in Figure 1b, which illustrates a finite state machine marked with the **SA Find And Pick task** ("SA" stands for a single agent), only one agent is allowed to execute this plan at the same time.
3.2 UPPAAL

UPPAAL is a toolbox for simulation, modeling, and verification of real time systems developed by Uppsala University in Sweden and Aalborg University in Denmark [4]. It uses an extended version of the timed automata formalism. Here, a model may include several automata that run concurrently with the capability of communicating via a number of channels, e.g., bounded integers.

Plans used in ALICA will be checked using UPPAAL, where three main components play their role in the process: a description language, a simulator and a verifier. The description language is a non-deterministic guarded command language with data types (e.g., channels and arrays). The simulator permits step by step simulation of the model displaying possible transitions and the current state. Finally, the verifier allows the specification and verification of properties as well as configuring the model checking engine. Also, if the verification attempt fails, a diagnostic trace may be loaded into the simulation component to investigate the issues.

4 MAPPING ALGORITHM AND A2U TOOL TRANSLATOR

In this section we describe the algorithm ALICA2UPPAAL, and based on it, an A2U tool translator is created to perform automatic translation of ALICA plans into UPPAAL. Initially, an overview of a mapping process is explained, followed by an explanation of steps conducted in the algorithm ALICA2UPPAAL.

Figure 3 shows an overview of the mapping process from ALICA plans to the UPPAAL model checker. The tool comprises the following functions:

1. **Input**: retrieving ALICA plan design in PML file format which contains the abstract syntax tree to store the data elements and transition relations in a plan.
2. **Parse**: the core component of the mapping process that maps elements of an ALICA plan to be used in UPPAAL. The data elements in the PML file are read and analyzed iteratively. Then, the C++ Parser creates three tables, where two of them have a fixed number of columns (i.e., locations and transitions). The last one is a dynamic table called a system.
3. **Output**: UPPAAL supports three file formats: XML, XTA, and TA. The XML is the newest format, supporting all features of UPPAAL e.g., anonymous locations are supported and graphical information (coordinates) about the objects are stored in XML file (in XTA format, graphical information (coordinates) are stored in a separate UGI file) and the PML and XML file have many similar properties. By these reasons, we choose XML format to be our output file.

The core contribution of this work is the Parse function. This process is divided into two steps: in the first step shown in Figure 4, it checks whether an ALICA plan can be converted to UPPAAL formalism. If it is not possible to do that, the process will stop. Otherwise, the Parse function whose flowchart depicted in Figure 5 will be executed. Here, the algorithm for mapping ALICA plans to UPPAAL, namely ALICA2UPPAAL is run through the following stages:

1. ALICA transitions are mapped into UPPAAL transitions. The property InState of an ALICA transition is converted to a corresponding transition Source (see Table 1). Similarly, the property OutState is converted to transition Target in UPPAAL model checker. Moreover, signal transmissions among agents or between the agent and its devices in ALICA are performed through communication channels inserted into UPPAAL transitions.
2. The Parse function converts ALICA states to UPPAAL locations. Because of the similarity of the structure, several

---

1 The file format used to store ALICA plans
We tested our ALICA2UPPAAL using the example provided in A2U translator written in C++ to implement our algorithm, which created from the ALICA plan using A2U. Figure 1b and showed the result in Figure 6, which is automatically translated to UPPAAL formalism.

![Uppaal Model](image)

**Figure 4: The main function of ALICA2UPPAAL algorithm.**

Otherwise, it performs another attempt to pick the object. Whether each agent picks up successfully or not, each one needs at least three time units for moving to a next state.

We take state GetObj as an example for our mapping process. Here, the time limit for each agent to wait for the result from its sensor is represented using $c4a \leq 9$ as an invariant. Then, the limit for moving to the next state is expressed by $c4a \geq 3$ as a guard of the outgoing transition. In this context, guard means the condition for “firing” the transitions in UPPAAL program. Since the state GetObj contains a plantype, the incoming transition placed is represented by the one with the channel ChannelCheckStatus. This channel will send a message to the initial location of the template CheckStatus converted from a plan that is a member of the plantype CheckStatus. Finally, the branch transitions placed and dropped are expressed via labels “Ok?” and “NotOk?” on their branching outgoing transition, respectively. These transitions receive the signals from PickUpSuc and PickUpUnSuc location to an initial location CheckStatus, respectively. After translating the ALICA plan into UPPAAL model structure, the verification process can begin. Once we have configured our scenario in UPPAAL we can formally verify the plan with the following properties:

- **Property 1:** The model is deadlock free.
  
  This property confirms that no deadlocks will occur in our model for all possible paths (the definition of deadlock is taken from UPPAAL [4]):

  $A[]$ not deadlock

- **Property 2:** The agents can successfully pick up the objects.

  $E<> \text{CheckStatus.PickUpSuc}$

  The results and diagnostic traces given by UPPAAL are presented in Figure 7.

  We intentionally created an error in the ALICA plan by switching the placed for dropped transitions, after re-translating the plan

<table>
<thead>
<tr>
<th>ALICA</th>
<th>UPPAAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>States</td>
<td>Locations</td>
</tr>
<tr>
<td>States.Id</td>
<td>Locations.Id</td>
</tr>
<tr>
<td>States.Names</td>
<td>Locations.Names</td>
</tr>
<tr>
<td>States.EntryPoint</td>
<td>Locations.Init (Initial location)</td>
</tr>
<tr>
<td>States.Coordinate</td>
<td>Locations.Coordinate</td>
</tr>
<tr>
<td>PlanType</td>
<td>Template</td>
</tr>
<tr>
<td>States.InTransition</td>
<td>-</td>
</tr>
<tr>
<td>States.OutTransition</td>
<td>-</td>
</tr>
<tr>
<td>Transitions</td>
<td>Transitions</td>
</tr>
<tr>
<td>Transition.InState</td>
<td>Transitions.Source</td>
</tr>
<tr>
<td>Transition.OutState</td>
<td>Transitions.Target</td>
</tr>
</tbody>
</table>

**Table 1: The properties of states and transitions of ALICA after translating to UPPAAL formalism.**

The purposes of ALICA plans can be mapped into properties in UPPAAL. Figure 7 shows the UPPAAL properties and diagnosis trace, which correspond to the ALICA plan in Figure 1b. Obviously, the plans should be verified before applying them to real agents. Regarding the run time for the translation process, it is highly depending on the complexity of the plans. Therefore, we have built the A2U translator written in C++ to implement our algorithm, which can automatically translate ALICA plans into UPPAAL model.

5 EXPERIMENTAL RESULTS

We tested our ALICA2UPPAAL using the example provided in Figure 1b and showed the result in Figure 6, which is automatically created from the ALICA plan using A2U.

Our scenario uses the assumption that the agent will leave the base within zero to 10 time units. Then, whenever it identifies an object, it executes the PickUp behavior for 11 to 20 time units. The feedback obtained from the sensor owned by the agent regarding the success of the picking, requires from zero to nine time units. If the object is picked successfully, the agent carries it to the base.

The success of the picking, requires from zero to nine time units. Then, whenever it identifies an object, it executes the PickUp behavior for 11 to 20 time units. The feedback obtained from the sensor owned by the agent regarding the success of the picking, requires from zero to nine time units. If the object is picked successfully, the agent carries it to the base.
by A2U we get the UPPAAL model presented in Figure 8. We verified the receptive UPPAAL model with the Property 1 to check the absence of deadlock in the system. The verification result said that there is a deadlock in the given model. From the diagnostic trace given by UPPAAL, we found out that the deadlock occurred when
Figure 7: The results of verification for a single agent in UPPAAL.

Figure 8: UPPAAL model of modified a single agent plan.

The agent reached to state GetObj, it cannot receive the corresponding signal from its sensor in order to move to the next state. The initial test shows that UPPAAL is able to verify the given properties, indicating its usability for ensuring correctness by detecting inconsistencies and flaws in ALICA plans. In the provided case, UPPAAL is able to detect a reachability property and the absence of deadlocks, which are useful for recognizing an incomplete specification of compatibilities in ALICA plans. Moreover, from checking the reachability of states and deadlock freedom, livelock properties in ALICA plans can also be detected.

To get a deeper impression on the results, we translate a second scenario in which the agents can collaborate if necessary. This plan is designed for multi-agents. Hence, the task must have an associated cardinality of \(2 \ldots \infty\). However, in this example, the maximum cardinality is five, as seen in Figure 9, because so far when we try to verify with five agents, some of properties would highly increase the verification time. This problem will be discussed at the end of this chapter. For this example, the mobile agents can perform following activities:

1. **Drive**: going out of the base and searching the selected objects.
2. **PickUp**: picking up the objects, if the agent has failed to pick them up, it sends a message to other agents and changes to Wait state.
3. **CheckStatus**: checking the feedback from a sensor that will inform the agent whether the pickup process is successful or not.
4. **PickUpTogether**: collaboratively by picking up the intended object (in the experiments a success rate at 100% is used).

After applying the A2U to the ALICA plans, we receive the UPPAAL model. Similar to the first scenario, we use the Property 1 to check the absence of deadlock in the system.

There is a deadlock in the given model, as shown by a red circle and the corresponding diagnostic traces given by UPPAAL in Figure 10. Note that Multiagent_Error and CheckStatusMA refer to UPPAAL templates, followed by a number in a pair of parentheses that refers to the identity of the agents and their respective sensor. From the diagnostic trace, we can observe that the deadlock occurred when two agents failed to pick up the objects, and both sent a message to the other agents and waited. To overcome this issue, we arrange a solution by adding one transition to the ALICA plan (from WaitOtherAgent to Committed state, see Figure 11). This transition allows the agents to come to help another one when all of them failed to pick up the objects. From the modified ALICA plan, we re-translate it into UPPAAL, as shown in Figure 12. After applying those changes, the plan is verified with the following properties (Property 1 and 2 are similar to the first scenario):

Property 3: The agents can collaborate to pick up the objects. Assume that we have released only two agents to work.

\[ E<> (\text{MultiAgent}\(0\).\text{PickUpTogether} + \text{MultiAgent}\(1\).\text{PickUpTogether}) = 2\]

Property 4: The propagation of a message from the agent to the sensor and vice versa happens in a bounded amount of time.

\[ A\[] (\text{MultiAgent}\(0\).\text{GetObj} \implies \text{CheckStatusMA}\(0\).\text{c4S} < 9)\]

Property 5: The agent will eventually come to pick with another one, if one of them fails to pick up the objects.

\[ A<> (\forall i:id.t \text{Status}[i] = 0) \implies \forall k:id.t \text{MultiAgent}(k).\text{PickUpTogether}\]

The verification results are shown in Figure 13. In our experiment, we recognize that in order to be verified, the properties require a substantial amount of processing time, particularly, when we increase the number of agents. For example, when there are only four agents working, the verification process for Property 4 takes around 45s to 47s seconds. However, when five agents are
Table 2: The verification time of UPPAAL with the different number of agents.

<table>
<thead>
<tr>
<th>Property</th>
<th>#Agents</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.001s</td>
<td>0.073s</td>
<td>0.554s</td>
<td>20.155s</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>2</td>
<td>0.001s</td>
<td>0.001s</td>
<td>0.003s</td>
<td>0.09s</td>
<td>0.027s</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>0.008s</td>
<td>0.01s</td>
<td>0.612s</td>
<td>3.143s</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>0.062s</td>
<td>1.2s</td>
<td>45.79s</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>0.001s</td>
<td>0.001s</td>
<td>0.001s</td>
<td></td>
<td>0.001s</td>
</tr>
</tbody>
</table>

- The time in the table is "elapsed time" given by UPPAAL model checker.
- Denote "*" to indicate that the verification did not terminate within a few hours and run out of memory.
- Denote "-" to indicate that those properties are not verified.

Figure 10: The verification result and diagnostic traces of multi-agents.

Figure 11: The ALICA plan of multi-agents after fixing.

Figure 12: The UPPAAL model of modified multi-agents after translating.

hours. Table 2 shows the amount of time required for verifying the properties.

6 CONCLUSION

This paper focuses on introducing the use of model checking for verifying ALICA plans. We presented an algorithm namely ALICA2UPPAAL that maps our ALICA plans into timed UPPAAL automata. Then, the technique is implemented using the A2U tool, which translates the plans automatically into UPPAAL syntax. Each
property will be examined in order to detect any inconsistencies and incompleteness within the model. Our proposed solution works well in translating a number of models having limited size and complexity, as demonstrated using two example scenarios.

We are currently working on several extensions for A2U, adding the capability to handle plans with higher size and complexity that are implemented in our soccer robots and other agents using the ALICA language. Furthermore, we are composing a database that will play the role as a "dictionary" which will make A2U convert ALICA plans into UPPAAL formalism quicker and more reliable. Last but not least, the A2U will be presented as an add-on to the graphical ALICA Plan Designer, so the user can easily check the current model in the middle of the building process.

ACKNOWLEDGMENTS

This work has been supported by a collaboration project funded by DFG/Germany under grant number GE776/20-1 and NAFOSTED/Vietnam under grant number DFG 102.01-2016.03.

REFERENCES


