Abstract. Unmanned aerial vehicles (UAV) are becoming increasingly popular for both recreational as well as industrial applications, leading also to concerns about safety. Autonomous systems, such as UAVs, are typically hybrid systems consisting of a low-level continuous control part and a high-level discrete decision making part. In this paper, we discuss using the agent programming language AgentSpeak to model the high-level decision making part. We present a translation from AgentSpeak to C that bridges the gap between high-level decision making and low-level control code for safety-critical systems. This allows code to be written in a more natural high-level language, thereby reducing the overall complexity and making it easier to maintain, while still conforming to safety guidelines. As an exemplar, we present the code for an autopilot for a UAV. The generated code is evaluated on a simulator and a Parrot AR.Drone, demonstrating the flexibility and expressiveness of AgentSpeak as a modeling language for UAVs.

1 Introduction

Unmanned Aerial Vehicles (UAV) have become an area of significant interest in the robotic community. This is mainly due to their capability to operate not only in open outdoor spaces, but also in confined indoor spaces, thus catering to a wide range of applications [33,17,24].

Current UAVs are usually remotely controlled by human pilots but partial or even full autonomy is a likely scenario in the foreseeable future, leading to various regulatory questions [21]. An autonomous UAV can be seen as a hybrid system consisting of a low-level continuous control part and a high-level decision making part [10,16]. While, arguably, the low-level control enables autonomy, it is the high-level decision processes that make the system truly autonomous. Owing to their unrestricted movement and usage of non-segregated airspace, safety is a primary concern for UAVs. As outlined in [10,16], questions and procedures relating to the safety of the low-level control are a well-established field. In this paper, we thus address the issue of the safety of autonomous decision making in UAVs.

Safety certification for aerial vehicles is guided by DO-178C – “Software Considerations in Airborne Systems an Equipment Certification” [31]. One particular
requirement is coding standards such as MISRA C \cite{misra}, which are routinely applied to low-level control code. While it is possible to implement the high-level decision making software directly in C, such code would, arguably, be hard to maintain as the language does not provide an adequate level of abstraction, which in turn also complicates establishing the various traceability requirements by DO-178C.

Agent-programming languages \cite{agent,agentspeak,agent-lang}, such as AgentSpeak \cite{agentspeak-lang}, were proposed as a solution for modeling high-level decision making. However, such languages often require an interpreter or similar run-time environment \cite{agent-lang}, for example Jason \cite{agent-lang} in the case of AgentSpeak. Using a setup like \cite{agent-lang} allows the high-level decision making to be interfaced with the low-level control code, however, it raises various concerns with regards to safety. The additional complexity of the system not only increases the number of potential sources of failure and required resources, but can also make traceability requirements infeasible, as not only actual agent code is affected, but also the run-time environment which, for example, in the case of Jason also includes the Java Virtual Machine.

In order to mitigate these problems, we propose to instead translate the high-level decision making code from the agent-programming language into the language of the low-level control code. This reduces the complexity of the resulting system and enables the reuse of already existing tools and processes for validation of the low-level control code. Furthermore, the traceability of requirements is guaranteed due to the systematic nature of the code translation. Lastly, but equally importantly, this allows the actual development to take place in the more natural setting of the agent-programming language, thus improving code maintainability.

This paper makes the following contributions.

– We present an automatic translation from AgentSpeak into a fragment of C akin to MISRA C for implementing high-level decision making in UAVs.
– As an exemplar, we present a re-implementation of the autopilot of the \texttt{tum_ardrone} package \cite{tum_ardrone} in AgentSpeak and show how the translated code can be used on a Parrot AR.Drone.

After briefly introducing the autopilot and AgentSpeak in Sections \ref{sec:autopilot} and \ref{sec:agentspeak}, we give an overview of the translation in Section \ref{sec:translation}. In Section \ref{sec:setup} we present our experimental setup, and finally in Section \ref{sec:safety} we discuss how this translation is an initial step to extend safety certification for the high-level decision making in UAVs and other autonomous systems.

\section{The Autopilot}

Automatic flight control systems are used to maintain given flight dynamic parameters and augment the stability of the system \cite{autopilot}. In order to achieve these

\footnote{While these requirements were originally applied to fixed-wing aircrafts, these high-level aspects translate directly to rotary-wing aircrafts.}
goals, typically some form of feedback control loop mechanism is employed, e.g., a PID (Proportional-Integral-Derivative) controller. While such a mechanism is clearly a low-level control aspect, one can also discern high-level decision making aspects in an autopilot, in particular the breaking down of the commands and coordinating their execution. One might also imagine a more sophisticated version of an autopilot implementing emergency landing upon low battery status, or collision avoidance, as even more prominent high-level aspects.

In the following, we will consider a simple autopilot for an unmanned quadrotor system. This autopilot, which we use as an exemplar, is based on the autopilot provided with the tum_ardrone package [12] (see also Section 5, however, our version only includes essential aspects). The autopilot supports three basic commands, namely takeoff, goto, and land. The takeoff command engages the rotors and brings the aircraft to a pre-defined altitude before handling any further commands. The goto command allows the specification of a target (in x, y, z and yaw coordinates) and the autopilot then moves the aircraft towards the target until it is reached. The land command lowers the aircraft to the ground and then disengages the rotors. Furthermore, if no commands are given by the user, the autopilot is required to maintain the aircraft’s current position.

In the following, we will show how the high-level aspects (specifically, the part called the “KI procedures” in the original tum_ardrone package) of this autopilot can be modeled in AgentSpeak, and how this AgentSpeak code can be translated in a way that ensures software considerations for safety critical systems.

3 AgentSpeak

AgentSpeak [26] is an agent-oriented programming language [31] based on the BDI (Belief– Desire–Intention) model [27,28]. Thus, in AgentSpeak, the central notion is that of an agent consisting of beliefs and plans. Agents react to events by selecting a plan, where plan selection is guided by the type of event and the current belief state of the agent. Once a plan is selected, it is executed, which can in turn create new events, change an agent’s beliefs, or execute basic actions available to the agent. Events can stem from external sources, i.e., the environment, or internal sources, i.e., the agent itself.

In order to illustrate these concepts, let us consider the example given in Figure 1, which gives a (simplified) fragment of an autopilot implemented in AgentSpeak. It gives the plans for an event corresponding to an instruction to go to a given position.

A typical sequence of execution would look as follows, assuming the UAV has already taken off and is airborne. An external source, for example the user, creates a +!goto event with a given target. In this case the first plan is selected, which subsequently updates the agent’s beliefs about the last set target and then the agent tries to achieve the goal completeGoto, which in turn creates the (internal) event +!completeGoto.
/* receive goto command, set target */
+!goto(Target) : takenOff
<- +lastTarget(Target);
  !completeGoto.

/* complete goto, check if already there, if not, move towards target */
+!completeGoto : takenOff & myPosition(Pos) & lastTarget(Target) & not closeEnough(Pos, Target)
<- Movement = calculateMovement(Pos, Target);
  sendControl(Movement);
  !completeGoto.

/* complete goto, check if already there, if yes, done */
+!completeGoto : takenOff & myPosition(Pos) & lastTarget(Target) & closeEnough(Pos, Target)
<- notifyUser("target reached").

Fig. 1. Plans for the goto case of the autopilot in AgentSpeak.

For this new event, the latter two plans are relevant. If the UAV’s current position is close enough to the given target, the third plan is applicable, in which case a notification to the user is issued (using a basic action provided by the environment) and nothing else remains to be done, thus leaving the autopilot ready for handling further events.

If the current position is not close enough to the target, the second plan is applicable. Here, the agent uses two basic actions in order to calculate the necessary control command and send it to the UAV. Finally, it tries to achieve the goal completeGoto again, which in turn creates a corresponding event and leads to the execution of either of the two plans until the target is reached.

Syntactically, we use a variant similar to AgentSpeak(F) as presented in [2,3], also assuming an explicitly given bound on the length of the event queue. For convenience, we add assignments as possible formulae. Another addition is “new focus” goals !!literal, which prove to be useful with the adopted semantics as explained below. The complete grammar is given in Table 1, where atoms are predicates or names of basic actions.

We impose one additional restriction that will become clear in view of the translation target (see Section 4). Roughly speaking, we only allow a restricted form of “recursion”. In order to explain this, we briefly need to introduce the following notion. A literal is called triggering, if it occurs as the main literal of the triggering event of a plan, else it is called non-triggering. In the body of a plan, we allow goals and percepts based on triggering literals only to occur as the last formula, any other goal or percept has to be based on a non-triggering literal.

For an overview of the semantics of AgentSpeak, see [36,6]. The reasoning cycle of an AgentSpeak agent works as follows. First, an event to be handled is selected. For this event, all plans that are relevant, i.e., those plans whose
triggering event unifies with the selected event, are found. This selection is then narrowed down to applicable plans, i.e., those plans whose context evaluates to true given the agent’s current beliefs. Finally, from these applicable plans one plan\(^2\) is selected to be actually executed, thus creating a so-called intention. Then, from all the intentions an agent currently has, one is selected to be executed, which means executing the next formula in the plan body and storing the updated intention accordingly.

The AgentSpeak architecture allows various customization options, see \[36\], in particular, belief revision, event selection, and intention selection can all be customized. Using these options, again in view of the translation target, we customize the semantics as follows:

- Events are handled in “run-to-completion” style, i.e., once a plan for a given event is selected, this plan is run until completed, which also involves any sub-plans triggered. This behavior can be achieved by using customized event and intention selection functions.

Note that new focus goals \(!!\text{literal}\) can be used in order to generate events that do not need to be handled immediately, i.e., that are not required to run to completion. Consider for example Figure 2 where this is used to implement a default behavior in case no user commands are given.

- The belief base of an agent can store at most one instance of a literal, i.e., if an agent holds the belief \(\text{speed}(3)\), after perpect \(+\text{speed}(5)\), the belief base is

\[\text{Table 1. AgentSpeak syntax.}\]

\[
\begin{aligned}
\langle \text{agent} \rangle &::= \langle \text{beliefs} \rangle \langle \text{initial\_goal} \rangle \langle \text{plans} \rangle \\
\langle \text{beliefs} \rangle &::= \langle \text{literal} \rangle \ldots \langle \text{literal} \rangle . \\
\langle \text{initial\_goal} \rangle &::= \langle \text{literal} \rangle . \\
\langle \text{plans} \rangle &::= \langle \text{plan} \rangle \ldots \langle \text{plan} \rangle \\
\langle \text{plan} \rangle &::= \langle \text{triggering\_event} \rangle : \langle \text{context} \rangle \leftarrow \langle \text{body} \rangle . \\
\langle \text{triggering\_event} \rangle &::= \langle \text{percept} \rangle \mid \langle \text{goal} \rangle \mid \langle \text{assignment} \rangle \\
\langle \text{context} \rangle &::= \langle \text{condition} \rangle \& \ldots \& \langle \text{condition} \rangle \\
\langle \text{condition} \rangle &::= \langle \text{literal} \rangle \mid \langle \text{not} \langle \text{literal} \rangle \rangle \\
\langle \text{body} \rangle &::= \langle \text{true} \rangle \mid \langle \text{formula} \rangle ; \ldots ; \langle \text{formula} \rangle \\
\langle \text{formula} \rangle &::= \langle \text{action} \rangle \mid \langle \text{percept} \rangle \mid \langle \text{goal} \rangle \mid \langle \text{assignment} \rangle \\
\langle \text{action} \rangle &::= \langle \text{literal} \rangle \\
\langle \text{percept} \rangle &::= \langle \text{not} \langle \text{literal} \rangle \rangle \mid \langle \text{goal} \rangle \\
\langle \text{goal} \rangle &::= \langle \text{not} \langle \text{literal} \rangle \rangle \mid \langle \text{not} \langle \text{literal} \rangle \rangle \mid \langle \text{assignment} \rangle \\
\langle \text{assignment} \rangle &::= \langle \text{variable} \rangle = \langle \text{literal} \rangle \\
\langle \text{literal} \rangle &::= \langle \text{atom} \rangle (\langle \text{term} \rangle , \ldots , \langle \text{term} \rangle ) \\
\langle \text{term} \rangle &::= \langle \text{variable} \rangle \mid \langle \text{unnamed variable} \rangle \mid \langle \text{number} \rangle \mid \langle \text{string} \rangle
\end{aligned}
\]
updated to speed(5), and will not contain an additional speed(3). This can be achieved using an appropriate belief revision function.

The purpose of these restrictions will be further illuminated in the following section. Note that even though we are only considering a fragment of AgentSpeak, it already suffices to model interesting processes, such as the autopilot.

4 Translation

We generally follow the ideas from [2,4,5], which present a translation from AgentSpeak to Promela [19] for the purposes of model checking AgentSpeak. As our intention is to run the generated code directly on the platform without any intermediate interpreters, our target language is C. As previously mentioned, software considerations for airborne systems are regulated by DO-178C [31]. While DO-178C does not prescribe the usage of any particular set of coding guidelines, MISRA C [22], or very similar rulesets, has become de-facto standard for safety-critical embedded software. We thus aim for our generated code to comply with the rules imposed by this standard, which prohibits recursion and dynamically allocated memory.

The syntactic restrictions and semantic customizations as introduced in the previous section directly relate to these restrictions. The restriction on “recursion” and the “run-to-completion” semantics give us a limit on the depth of the call stack, eliminate the need for handling and storing multiple intentions, and, furthermore, also limit the number of events generated internally, thus allowing for a fixed-length event queue. Our custom belief revision allows us to model the belief base using one single variable per literal (plus an additional flag that indicates whether it is set or not), again not requiring dynamic memory.

Due to space limitations it is not possible to give the complete translation algorithm [8] but the following examples should provide all relevant ideas. At the core of the translated code lies the `next_step()` function corresponding to one step in the agent’s reasoning cycle. First of all, the hook `updateBeliefs()` allows the environment to update the agent’s belief state, if required. Then, the next event is dequeued and a relevant plan is selected, as illustrated in Figure 3. Subsequently, the first applicable plan (in order of textual appearance) is selected and executed, see Figure 4 for a concrete instance. Note that this would also allow for the inclusion of error handling in case no plan is applicable (cf. [6]).

3 The full code is available at [8].

Fig. 2. One plan for the `waitForCommand` case for the autopilot in AgentSpeak.
void next_step(void) {
    updateBeliefs();
    eventt event = get_next_event();
    switch (event.trigger) {
        /* ... */
        case ADD_ACHIEVE_GOTO:
            add_achieve_goto(event.goto_param0); break;
        case ADD_ACHIEVE_COMPLETEGOTO:
            add_achieve_completeGoto(); break;
        /* ... */
    }
}

void add_achieve_completeTakeOff(int param0) {
    if (add_achieve_completeTakeOff_plan0(param0)) { return; }
    if (add_achieve_completeTakeOff_plan1(param0)) { return; }
    /* ... handle the case where no plan is applicable ... */
    return;
}

The translation of the plans themselves is illustrated in Figures 3 and 4. First, a plan tests for its own applicability, instantiating variables on the way, if necessary. Basic actions and assignments translate directly to their C counterparts. Percepts set or unset the corresponding variables in the belief base, and test goals read the corresponding variable on belief bases.

Achievement goals require a more elaborate handling, distinguishing three different cases. Non-triggering achievement goals call their relevant plan selection method directly. Triggering achievement goals create a new event and insert it at the head of the event queue, thus guaranteeing it will be handled next and enforcing the run-to-completion semantics. For example, this is can be seen as internal_achieve_completeGoto(); at the end of the code in Figure 5. New focus achievement goals also create a new event and insert it at the tail of the event queue, thus allowing for other events to be handled before them. An example for this is achieve_waitForCommand(); at the end of the code in Figure 6. Note that external events are also added at the tail of the event queue, however, one might consider a variant where external events can pre-empt other events by having them added at the head of the queue.

Besides the elements presented above, data structures and code for handling the event queue and belief base are also generated. As this is done in the obvious manner, we have omitted it from the current presentation.
bool add_achieve_goto_plan0(positiont param0) {
    positiont Target = param0;
    if (!takenOff_set) { return false; }
    sendHover();
    lastTarget_set = true;
    lastTarget_param0 = Target;
    internal_achieve_completeGoto();
    return true;
}

Fig. 5. Translation of the goto plan from Figure 1 into C.

bool add_achieve_waitForCommand_plan1(void) {
    if (!takenOff_set) { return false; }
    if (!lastTarget_set) { return false; }
    positiont Target = lastTarget_param0;
    if (!myPosition_set) {
        /* ... achieve test goal or handle plan failure ... */
    }
    positiont Pos = myPosition_param0;
    control_commandt Movement = calculateMovement(Pos, Target);
    sendControl(Movement);
    achieve_waitForCommand();
    return true;
}

Fig. 6. Translation of the waitForCommand plan from Figure 2 into C.

5 Experimental Setup

We evaluated the generated code on the tum_simulator package [20] and a Parrot AR.Drone 2.0 using the Robot Operating System (ROS) [25,30] version Hydro Medusa with the tum_ardrone package [12,14,13,15]. The availability of the tum_simulator facilitated rapid prototyping and development of the translator, as translated code could be easily tested without the need of an extensive experimental setup. Due to the modular nature of ROS, switching to the actual platform was a straightforward operation.

It is important to note that the generated code is not specific to ROS. Thus, it is necessary to provide wrapper code acting as an interface between ROS and the generated code, translating user commands into AgentSpeak events and providing the basic actions required by the AgentSpeak code. The original tum_ardrone package already uses a similar arrangement, consisting of three components, namely the autopilot node, a PID controller for low-level control, and so-called “KI procedures” for the high-level control. For our experiments,

While it would be possible to generate ROS code directly, we chose a more general solution such that the generated code can also be used in settings that are not based on ROS.
we replaced these KI procedures with our generated code and we modified the autopilot node accordingly. The complete setup is illustrated in Figure 7. In addition to the autopilot node, a simple user interface for the input of flight plans is provided. Furthermore, a state estimation node is used to track the current position of the UAV based on navigation data and the video feed. We have tested various flight plans composed of the three basic commands takeoff, goto and land. We compared our autopilot to the original tum_ardrone autopilot and have confirmed analogue behavior. A screenshot of a simulated flight is depicted in Figure 8.

On the technical side, the AgentSpeak code for the autopilot uses about 50 lines of code and generates about 500 lines of C code compared to the about 300 lines of C++ code of the original KI procedures. This shows that AgentSpeak allows a much more compact representation of the high-level behavior of the autopilot, making the code easier to maintain and extend. The translator code as well as the modified tum_ardrone package are available online [8], where detailed instructions regarding installation are also provided.

6 Conclusions and Future Work

UAVs can be seen as autonomous agents consisting of low-level control and high-level decision making. Making changes to both of these parts can be a challenging task and is prone to errors. Since UAVs have to comply with strict safety standards, changing the low-level and high-level parts must be done without introducing any errors that could potentially lead to safety issues.
In this paper, we discuss using AgentSpeak as a modeling language for the high-level decision making of UAVs. This model can then be automatically translated into C, thus bridging the gap between the high-level decision making and the low-level control code. The abstract model in AgentSpeak reduces the complexity of the code and is flexible for maintenance and further extensions. The automatic translation removes possible errors that can be introduced when mixing these high-level aspects directly into the low-level control code and complies with the safety regulations for UAVs. As an example, we show how the autopilot of the tum_ardrone package can be easily modeled in AgentSpeak and how the generated C code can be directly used in real world platforms, such as a Parrot AR.Drone. We also remark that this approach is not restricted solely to UAVs, but can be adapted to various other autonomous systems.

As future work, we plan to expand the translator to cover richer fragments of AgentSpeak, thus allowing the modeling of more complex operations for UAVs. Safety certification is an essential criteria for the practical usage of UAVs. To guarantee safety, we plan to verify safety properties of the AgentSpeak model by implementing the operational semantics in term rewrite systems like [29,11]. We also plan to validate the translation from AgentSpeak to C code with the assistance of CBMC [9], a bounded model checker for C. Translation validation [32,35] is a common approach to guarantee that the semantics of the high-level model are preserved in the translated code. Overall, this will guarantee that the safety properties established for the AgentSpeak model can be transferred to the translated code, thereby guaranteeing the required traceability of requirements.

Acknowledgments This work was supported by Engineering and Physical Sciences Research Council (EPSRC) grant EP/J012564/1 “New Foundational Structures for Engineering Verified multi-UAVs”.

Fig. 8. A screenshot of the autopilot user interface and map view.
References


