From AgentSpeak to C for Safety Considerations in Unmanned Aerial Vehicles

Samuel Bucheli(✉), Daniel Kroening, Ruben Martins, and Ashutosh Natraj

Department of Computer Science, University of Oxford, Oxford, UK
{samuel.bucheli,daniel.kroening,ruben.martins,
ashutosh.natraj}@cs.ox.ac.uk
http://www.cprover.org

Abstract. Unmanned aerial vehicles (UAV) are becoming increasingly popular for both recreational and industrial applications, leading to growing 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. 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 its overall complexity and making it easier to maintain, while still conforming to safety guidelines. As an exemplar, we present the code for a UAV autopilot. 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 (UAVs) 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 [27].

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 [9,15]. 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 UAV use. As outlined in [9,15], questions and procedures relating to the safety of the low-level control are well-established. 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 [34]. One particular requirement is coding standards such as MISRA C [23], which are routinely

C. Dixon and K. Tuyls (Eds.): TAROS 2015, LNAI 9287, pp. 69–81, 2015.
DOI: 10.1007/978-3-319-22416-9_9

applied to low-level control code. While it is possible to implement the high-level decision making 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 [2,39,44], such as AgentSpeak [29], were proposed as a solution for modeling high-level decision making. However, such languages often require an interpreter or similar run-time environment [2], for example Jason [5] in the case of AgentSpeak. Using a setup like [17] 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.

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. This allows the actual development to take place in the more natural setting of the agent-programming language, thus improving code maintainability.

To the best of our knowledge, no such approach has been proposed for AgentSpeak so far. Related work includes the automated code generation from behavior-based agent programming with the aim of facilitating development and code reuse [43], the translation of AgentSpeak to Promela proposed in [3] for the purpose of model checking AgentSpeak, and the translation of AgentSpeak to Erlang in order to improve concurrent performance [11]. In a wider context, the usage and translation of statecharts (e.g. [18,22]) and Stateflow (e.g. [1,38]) for modelling and the relation between statecharts and agent-oriented programming [16] provides further background and inspiration to this work.

This paper makes the following contributions.

– We present an automated 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 `tum_ardrone` package [13] in AgentSpeak and show how the translated code can be used on a Parrot AR.Drone.

2 The Autopilot

Automatic flight control systems are used to maintain given flight dynamic parameters and to augment stability [24].[1] In order to achieve these 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

[1] While these requirements were originally applied to fixed-wing aircrafts, these high-level aspects translate directly to rotary-wing aircrafts.

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 collision avoidance, or an emergency landing upon low battery status, 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 [13], however, our version only includes essential aspects (see also Section 5). 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 we model the high-level aspects (specifically, the part called the "KI procedures" in the original `tum_ardrone` package) of this autopilot in AgentSpeak, and how we translate this code in a way that ensures software considerations for safety critical systems.

3 AgentSpeak

AgentSpeak [29] is an agent-oriented programming language [39] based on the BDI (Belief-Desire-Intention) model [30,31]. Thus, in AgentSpeak, the central notion is that of an agent consisting of beliefs and plans. Agents react to events by selecting a plan, and 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`.

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.

```
+!goto(Target) : takenOff
  <- sendHover(); +lastTarget(Target);
     !completeGoto.

+!completeGoto : takenOff & myPosition(Pos) & lastTarget(Target)
                   & not closeEnough(Pos, Target)
  <- Movement = calculateMovement(Pos, Target);
     sendControl(Movement);
     !completeGoto.

+!completeGoto : takenOff & myPosition(Pos) & lastTarget(Target)
                   & closeEnough(Pos, Target)
  <- sendHover(); notifyUser("target reached").
```

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

```
+!waitForCommand : takenOff & lastTarget(Target)
  <- ?myPosition(Pos);
     Movement = calculateMovement(Pos, Target);
     sendControl(Movement);
     !!waitForCommand. // new focus!
```

Fig. 2. One plan for the `waitForCommand` case for the autopilot in AgentSpeak

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 [3]. 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 notions. Similar to a call graph, we can create a *trigger graph* of a given set of plans indicating which plans may trigger the execution of which further plans. Note that as in the case of call graphs, this trigger graph is an over-approximation. We call an atom *recursively triggering*, if it occurs as the atom of the triggering event of a plan from which a cycle is reachable in the trigger graph. In the body of a plan, we allow goals and percepts based on recursively triggering atoms only to occur as the last formula; any other goal or percept has to be based on a non-recursively triggering atom.

For an overview of the semantics of AgentSpeak, see [4,42]. The reasoning cycle of an AgentSpeak agent works as follows. First, an event to be handled

Table 1. AgentSpeak syntax

⟨*agent*⟩	::=	⟨*beliefs*⟩ ⟨*initial_ goal*⟩ ⟨*plans*⟩
⟨*beliefs*⟩	::=	⟨*literal*⟩`.` …⟨*literal*⟩`.`
⟨*initial_ goal*⟩	::=	`!`⟨*literal*⟩ `.`
⟨*plans*⟩	::=	⟨*plan*⟩ …⟨*plan*⟩
⟨*plan*⟩	::=	⟨*triggering_ event*⟩ `:` ⟨*context*⟩ `<-` ⟨*body*⟩`.`
⟨*triggering_ event*⟩	::=	⟨*percept*⟩ | `+`⟨*goal*⟩ | `-`⟨*goal*⟩
⟨*context*⟩	::=	⟨*condition*⟩ `&` … `&` ⟨*condition*⟩
⟨*condition*⟩	::=	`true` | ⟨*literal*⟩ | `not(` ⟨*literal*⟩ `)`
⟨*body*⟩	::=	`true` | ⟨*formula*⟩`;` … `;` ⟨*formula*⟩
⟨*formula*⟩	::=	⟨*action*⟩ | ⟨*percept*⟩ | ⟨*goal*⟩ | ⟨*assignment*⟩
⟨*action*⟩	::=	⟨*literal*⟩
⟨*percept*⟩	::=	`+`⟨*literal*⟩ | `-`⟨*literal*⟩
⟨*goal*⟩	::=	`!`⟨*literal*⟩ | `!!`⟨*literal*⟩ | `?`⟨*literal*⟩
⟨*assignment*⟩	::=	⟨*variable*⟩ `=` ⟨*literal*⟩
⟨*literal*⟩	::=	*atom*`(` ⟨*term*⟩`,` … `,` ⟨*term*⟩ `)`
⟨*term*⟩	::=	*variable* | *unnamed variable* | *number* | *string*

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. From these applicable plans one plan[2] is selected to be actually executed, 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 [42]. In particular, belief revision, event selection, and intention selection can all be customized. In view of the translation target, we use the following options.

First, 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 `!!literal` can be used in order to generate deferred events, i.e., events that do not need to be handled immediately. Consider for example Figure 2, where this is used to implement a default behavior in case no user commands are given.

Second, the belief base of an agent can store at most one instance of a literal, i.e., if an agent holds the belief `speed(3)`, after percept `+speed(5)`, the belief base is updated to `speed(5)`, and will not contain an additional `speed(3)`. This can be achieved using an appropriate belief revision function.

[2] The default behavior is to select the first applicable plan in textual order.

```
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;
   /* ... */
  }
}
```

Fig. 3. Selection of relevant plans for the autopilot, translated to C

```
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;
}
```

Fig. 4. Selection of applicable plans, for the `completeTakeOff` case

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 [3], 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 [34]. While DO-178C does not prescribe the usage of any particular set of coding guidelines, MISRA C [23], 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 to at most one per deliberation cycle. Thus, we only need to store at most two events, one event

```
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);
  internal_achieve_new_focus_waitForCommand();
  return true;
}
```

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

that needs to be run to completion,[3] and, possibly, a deferred event. 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. Note also that the restriction on "recursion" gives natural break points to segment long runs to completion into smaller parts, thus ensuring the reactivity of the translated program.

The translation proceeds in three main stages. First of all, the AgentSpeak program and an accompanying configuration file are parsed. Then, the parsed program is analyzed, which includes determining the recursively triggering atoms. Finally, the actual translation takes place, which we will outline below.

Due to space limitations it is not possible to give all details,[4] 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 selected. Here, the rtc event is given preference over the deferred event. Then a relevant plan is

[3] We will call this the *rtc event* in the following.
[4] The full code is available at [6].

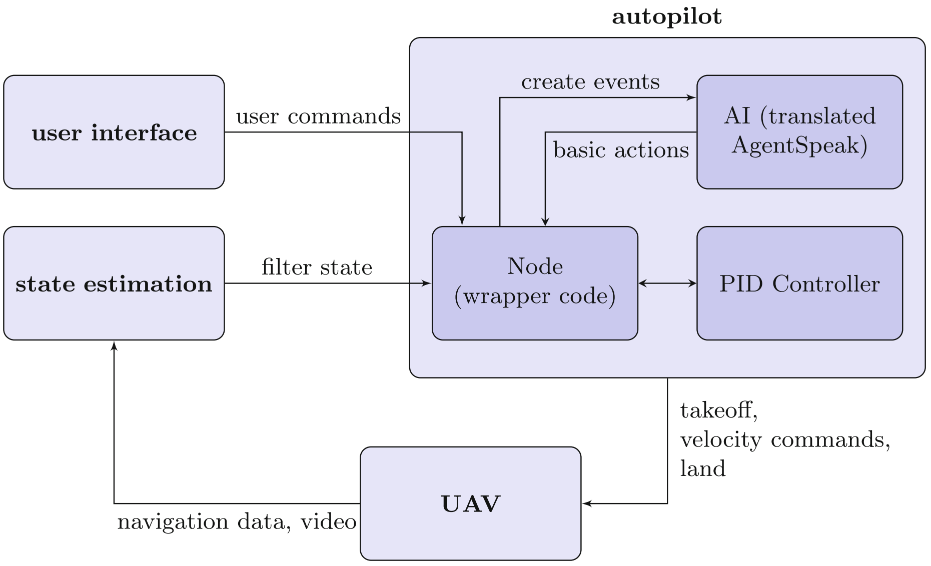

Fig. 7. Experimental setup

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 the inclusion of error handling if no plan is applicable (cf. [4]).

The translation of the plans themselves is illustrated in Figures 5 and 6. First, a plan tests for its own applicability, instantiating variables along 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-recursively triggering achievement goals call their relevant plan selection method directly. Recursively triggering achievement goals create a new rtc event, thus guaranteeing it will be handled next. New focus achievement goals create a deferred event, thus allowing for rtc events to be handled before them. Note that external events are created as rtc events and can only be created if there is currently no rtc event set. Otherwise, the creation of the external event has to wait until the current rtc event has finished. However, one might image more elaborate versions where certain external events can pre-empt internal rtc events.

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) [28,33] version

Hydro Medusa with the `tum_ardrone` package [13,14]. 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.[5] 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, 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.

On the technical side, the AgentSpeak code for the autopilot uses about 70 lines of code and generates about 700 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 modified `tum_ardrone` package is available online [7], where detailed instructions regarding installation are also provided.

6 Conclusions and Future Work

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.

[5] 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.

Future research includes identifying fragments of AgentSpeak that are more expressive while still allowing translation within the given restrictions (cf. [38]). This also includes adequate methods for static analysis and testing, as typically required for safety certification, an essential criteria for the practical usage of UAVs. Formal methods (cf. [35]) complement these approaches and we plan to verify safety properties of the AgentSpeak model by implementing the operational semantics in term rewrite systems like [12,32]. We also plan to validate the translation from AgentSpeak to C code with the assistance of CBMC [8], a bounded model checker for C. Translation validation [26,36,37,40] 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.

For example, consider [25] stating "[...] satisfying control of the AR.Drone 2.0 is reached by sending the AT-commands every 30 ms [...]".[6] Using [41], one could establish the worst-case execution time of the translated code. On the other hand, the statement "*in every deliberation cycle one of the aforementioned basic actions is executed*" can be easily formalized in linear temporal logic using BDI primitives. This property could then be checked on the AgentSpeak code using the approach mentioned above. Using the translation validation would then allow us to combine these facts, thus establishing the given requirement.

As a final note, the separation of high-level and low-level concerns allows reusing results on both ends of the translation. For example, one might use Jason [5] (with appropriate custom semantics) as a simulation environment for prototyping. Also, the formal verification on the AgentSpeak is not restricted to BDI properties, but, as outlined in [10] various other options, e.g., probabilistic properties can also be considered, thus allowing even more flexibility.

Acknowledgments. This work was supported by Engineering and Physical Sciences Research Council (EPSRC) grant EP/J012564/1.

References

1. Agrawal, A., Simon, G., Karsai, G.: Semantic Translation of Simulink/Stateflow Models to Hybrid Automata Using Graph Transformations. Electronic Notes in Theoretical Computer Science **109**, 43–56 (2004)
2. Bordini, R.H., Braubach, L., Dastani, M., Fallah-Seghrouchni, A.E., Gómez-Sanz, J., Leite, J., O'Hare, G., Pokahr, A., Ricci, A.: A Survey of Programming Languages and Platforms for Multi-Agent Systems. Informatica **30**(1), 33–44 (2006)
3. Bordini, R.H., Fisher, M., Visser, W., Wooldridge, M.: Verifying Multi-agent Programs by Model Checking. Autonomous Agents and Multi-Agent Systems **12**(2), 239–256 (2006)

[6] For our purposes, *AT-commands* can be identified with the basic actions for taking off, landing, and sending movements. Note that hovering is a form of movement.

4. Bordini, R.H., Hübner, J.F.: Semantics for the jason variant of agentspeak (plan failure and some internal actions). In: European Conference on Artificial Intelligence, pp. 635–640. IOS Press (2010)
5. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak Using Jason. John Wiley & Sons (2007)
6. Bucheli, S., Kroening, D., Martins, R., Natraj, A.: AgentSpeak Translator. https://github.com/OxfordUAVAutonomy/AgentSpeakTranslator (last visited June 15, 2015). doi:10.5281/zenodo.18572
7. Bucheli, S., Kroening, D., Martins, R., Natraj, A.: Modified `tum_ardrone` package (last visited 15, June 2015). doi:10.5281/zenodo.18571. https://github.com/OxfordUAVAutonomy/tum_ardrone
8. Clarke, E., Kroning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
9. Dennis, L.A., Fisher, M., Lincoln, N., Lisitsa, A., Veres, S.M.: Practical Verification of Decision-Making in Agent-Based Autonomous Systems. Automated Software Engineering Online, 1–55 (2014). doi:10.1007/s10515-014-0168-9
10. Dennis, L.A., Fisher, M., Webster, M.: Two-stage agent program verification. Journal of Logic and Computation Online (2015). doi:10.1093/logcom/exv002
11. Díaz, Á.F., Earle, C.B., Fredlund, L.Å.: eJason: an implementation of jason in erlang. In: Dastani, M., Hübner, J.F., Logan, B. (eds.) ProMAS 2012. LNCS(LNAI), vol. 7837, pp. 1–16. Springer, Heidelberg (2013)
12. Doan, T.T., Yao, Y., Alechina, N., Logan, B.: Verifying heterogeneous multi-agent programs. In: International Joint Conference on Autonomous Agents and Multiagent Systems. IEEE Computer Society (2014)
13. Engel, J., Sturm, J., Cremers, D.: `tum_ardrone`. http://wiki.ros.org/tum_ardrone (last visited June 15, 2015)
14. Engel, J., Sturm, J., Cremers, D.: Scale-Aware Navigation of a Low-Cost Quadrocopter with a Monocular Camera. Robotics and Autonomous Systems **62**(11), 1646–1656 (2014)
15. Fisher, M., Dennis, L., Webster, M.: Verifying autonomous systems. Communications of the ACM **56**(9), 84–93 (2013)
16. Fortino, G., Rango, F., Russo, W., Santoro, C.: Translation of statechart agents into a BDI framework for MAS engineering. Engineering Applications of Artificial Intelligence **41**, 287–297 (2015)
17. Hama, M.T., Allgayer, R.S., Pereira, C.E., Bordini, R.H.: UAVAS: AgentSpeak Agents for Unmanned Aerial Vehicles. In: Workshop on Autonomous Software Systems at CBSoft (Autosoft 2011) (2011)
18. Harel, D., Politi, M.: Modeling Reactive Systems with Statecharts: The Statemate Approach, 1st edn. McGraw-Hill Inc., New York (1998)
19. Holzmann, G.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley Professional (2003)
20. Huang, H., Sturm, J.: `tum_simulator`. http://wiki.ros.org/tum_simulator (last visited June 15, 2015)
21. ICAO Cir 328: Unmanned Aircraft Systems (UAS). International Civil Aviation Organization (2011)

22. Mikk, E., Lakhnech, Y., Siegel, M., Holzmann, G.: Implementing statecharts in PROMELA/SPIN. In: Workshop on Industrial Strength Formal Specification Techniques, pp. 90–101. IEEE Computer Society Press (1998)
23. Motor Industry Software Reliability Association and Motor Industry Software Reliability Association Staff: MISRA C: 2012: Guidelines for the Use of the C Language in Critical Systems. Motor Industry Research Association (2013)
24. Nelson, R.C.: Flight stability and automatic control. McGraw-Hill (1997)
25. Piskorski, S., Brulez, N., Eline, P., D'Haeyer, F.: AR.Drone Developer Guide Revision SDK 2.0. Parrot S.A. (2012)
26. Pnueli, A., Siegel, M.D., Singerman, E.: Translation validation. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 151–166. Springer, Heidelberg (1998)
27. Puri, A.: A Survey of Unmanned Aerial Vehicles (UAV) for Traffic Surveillance. Department of Computer Science and Engineering, University of South Florida, Tech. rep. (2005)
28. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: ROS: an open-source robot operating system. In: ICRA Workshop on Open-Source Software in Robotics (2009)
29. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: Perram, J., Van de Velde, W. (eds.) MAAMAW 1996. LNCS, vol. 1038, pp. 42–55. Springer, Heidelberg (1996)
30. Rao, A.S., Georgeff, M.P.: BDI Agents: From theory to practice. In: International Conference on Multi-Agent Systems, pp. 312–319. AAAI press (1995)
31. Rao, A.S., Georgeff, M.P.: Modeling rational agents with a BDI-architecture. In: Readings in Agents, pp. 317–328. Morgan Kaufmann Publishers Inc. (1998)
32. van Riemsdijk, M.B., de Boer, F.S., Dastani, M., Meyer, J.J.C.: Prototyping 3APL in the Maude Term Rewriting Language. In: International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 1279–1281. ACM (2006)
33. ROS: Robot Operating System. http://www.ros.org (last visited June 15, 2015)
34. RTCA: DO-178C, Software Considerations in Airborne Systems and Equipment Certification. Radio Technical Commission for Aeronautics (2011)
35. RTCA: DO-333, Formal Methods Supplement to DO-178C and DO-278A. Radio Technical Commission for Aeronautics (2011)
36. Ryabtsev, M., Strichman, O.: Translation validation: from simulink to C. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 696–701. Springer, Heidelberg (2009)
37. Sampath, P., Rajeev, A.C., Ramesh, S.: Translation validation for stateflow to C. In: Design Automation Conference on Design Automation Conference, pp. 23:1–23:6. ACM (2014)
38. Scaife, N., Sofronis, C., Caspi, P., Tripakis, S., Maraninchi, F.: Defining and translating a "safe" subset of simulink/stateflow into lustre. In: International Conference on Embedded Software, pp. 259–268. ACM (2004)
39. Shoham, Y.: Agent-oriented Programming. Artificial Intelligence **60**(1), 51–92 (1993)
40. Staats, M., Heimdahl, M.P.E.: Partial translation verification for untrusted code-generators. In: Liu, S., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 226–237. Springer, Heidelberg (2008)

41. Thesing, S., Souyris, J., Heckmann, R., Randimbivololona, F., Langenbach, M., Wilhelm, R., Ferdinand, C.: An abstract interpretation-based timing validation of hard real-time avionics software. In: International Conference on Dependable Systems and Networks, pp. 625–632. IEEE (2003)
42. Vieira, R., Moreira, A., Wooldridge, M., Bordini, R.H.: On the formal semantics of speech-act based communication in an agent-oriented programming language. Journal of Artificial Intelligence Research **29**(1), 221–267 (2007)
43. Vu, T., Veloso, M.: Behavior Programming Language and Automated Code Generation for Agent Behavior Control. School of Computer Science, Carnegie Mellon University, Tech. rep. (2004)
44. Ziafati, P., Dastani, M., Meyer, J.-J., van der Torre, L.: Agent programming languages requirements for programming autonomous robots. In: Dastani, M., Hübner, J.F., Logan, B. (eds.) ProMAS 2012. LNCS(LNAI), vol. 7837, pp. 35–53. Springer, Heidelberg (2013)