Practical Application of SPARK to OpenUxAS*

M. Anthony Aiello[1], Claire Dross[2], Patrick Rogers[1], Laura Humphrey[3], and James Hamil[4]

[1] AdaCore Technologies, Inc., New York NY 10001, USA
[2] AdaCore SAS, 75009 Paris, France
[3] Air Force Research Laboratory, Dayton OH 45433, USA
[4] LinQuest Corporation, Beavercreek OH 45431, USA

Abstract. This paper presents initial, positive results from using SPARK to prove critical properties of OpenUxAS, a service-oriented software framework developed by AFRL for mission-level autonomy for teams of cooperating unmanned vehicles. Given the intended use of OpenUxAS, there are many safety and security implications; however, these considerations are unaddressed in the current implementation. AFRL is seeking to address these considerations through the use of formal methods, including through the application of SPARK, a programming language that includes a specification language and a toolset for proving that programs satisfy their specifications. Using SPARK, we reimplemented one of the core services in OpenUxAS and proved that a critical part of its functionality satisfies its specification. This successful application provides a foundation for further applications of formal methods to OpenUxAS.

Keywords: OpenUxAS · SPARK · Formal Methods · Autonomy

1 Introduction

This paper presents initial, positive results from using SPARK to prove critical properties of OpenUxAS, a software framework for mission-level autonomy for teams of cooperating unmanned vehicles.

Efficient and effective use of unmanned vehicles requires greater levels of autonomy than employed today. Currently, command and control of a single vehicle requires multiple human operators to perform lower-level tasks such as path planning, piloting, sensor steering, and so forth. Automating these lower-level tasks would ideally allow multiple vehicles to be managed by a single operator, increasing efficiency and allowing the operator to focus on tactical and strategic aspects of the mission rather than low-level execution details. Toward this end, additional automation could build off of these tasks to provide the operator with a high-level interface to command and control multiple vehicles. Additionally, communication channels between vehicles are often unreliable, so services must function with only intermittent communication between vehicles.

* DISTRIBUTION STATEMENT A: Distribution unlimited; approved for public release; case number 88ABW-2017-1985.

The United States Air Force Research Laboratory (AFRL) has explored solutions to this problem through the research and development of decentralized cooperative control approaches [4]. AFRL has developed a service-oriented architecture called Unmanned Systems Autonomy Services (UxAS) that provides services handling many of the low-level details necessary for decentralized cooperative control and tasks[5] implementing high-level command and control, thus accelerating research and development in this area. (Although the main focus of UxAS is aircraft, the 'x' in UxAS indicates support for other vehicles.)

AFRL has created a public-release, open-source version of UxAS, called OpenUxAS, and has made OpenUxAS[6] and a compatible multi-vehicle simulation environment (OpenAMASE[7]) available on github. UxAS is implemented in C++ 11, but the messages used for communication between services are described using AFRL's language-neutral Lightweight Message Construction Protocol (LMCP), allowing UxAS tasks and services to be written in other languages. LMCP is also available on github[8].

Given the intended use of UxAS, there are many safety and security implications. Because UxAS was developed initially to accelerate internal research and development, these considerations are currently left unaddressed. However, as interest in UxAS grows, both within AFRL and also in the broader community, addressing the safety and security of UxAS becomes increasingly important.

The known limitations of testing [1], especially for autonomy, make the application of formal methods to UxAS a priority for AFRL. Because UxAS was originally intended to facilitate research, the design and implementation of UxAS does not always lend itself well to the application of formal methods. For example, in addition to being implemented in C++, UxAS makes frequent use of pointers and does not define application-specific ranges for numeric values.

AFRL and AdaCore have therefore collaborated to rewrite parts of OpenUxAS in Ada 2012 and SPARK 2014 so that SPARK can prove critical properties of core services. In this paper, we describe our implementation approach, present initial results, and identify our next objectives.

2 Background

2.1 OpenUxAS

UxAS is designed to be highly extensible and configurable: depending on the configuration of loaded services and tasks, which perform mission-specific activities such as area, line or point surveillance [3], UxAS can perform a variety of missions, including decentralized surveillance [4] or ground-intruder isolation. At the heart of UxAS is the task-assignment pipeline, which is implemented as

[5] For the remainder of the paper, we use "task" to refer to a component of a mission in UxAS (see: [3]). When we refer to an Ada task, we will clearly indicate it as such.
[6] https://github.com/afrl-rq/OpenUxAS
[7] https://github.com/afrl-rq/OpenAMASE
[8] https://github.com/afrl-rq/LmcpGen

a set of cooperating services [5]. The role of the task-assignment pipeline is to take tasks, with associated task orderings or dependencies, and distribute them amongst eligible vehicles. The goal of the distribution is to be time-optimal: all tasks should be completed by the eligible vehicles as quickly as possible. Services and tasks communicate by exchanging messages, defined using LMCP, over the message bus, which is implemented using ZeroMQ[9].

For this work, we focus on the Automation Request Validator service, which validates and serializes new Automation Request messages. Automation Request messages describe missions by referencing tasks, eligible vehicles, and operating regions by their IDs, which must have been previously defined by other received messages. The service thus acts as a gatekeeper, performing two functions: (1) ensure that the Automation Request can be carried out, by checking that the ID of every vehicle, task and operating region referenced has previously been defined; and then (2) ensure that only one resulting actionable request, in the form of a Unique Automation Request message, is fed into the rest of the system at a time. Our focus for the application of SPARK is on the first function.

2.2 SPARK

SPARK is both a programming language with a specification language and a toolset that is supported by specific development and verification processes [2]. Here, we focus on the latest generation of SPARK, SPARK 2014, in which the specification language and the programming language have been unified as a subset of Ada 2012. SPARK excludes features not amenable to sound static verification, principally access types (pointers), function side effects, and exception handling. Constraints on both program data and control can be specified using type contracts (predicates and invariants) and function contracts (preconditions and postconditions), respectively. The SPARK verification toolset can automatically prove that an implementation conforms to its specification and is free from run-time exceptions.

3 Approach

Our approach is to translate the Automation Request Validator service from C++ to Ada and SPARK. We use Ada to implement the message-based communication classes above ZeroMQ the object-oriented class hierarchy for the service classes. Both Ada and SPARK are used to implement the concrete Automation Request Validator service subclass. In particular, SPARK is used to implement the critical functionality of the service, i.e., the part that validates the Automation Request messages.

We follow the C++ design closely so that any errors encountered will not be due to a design change we introduced. Some changes are required for SPARK, and are described below. As noted in Section 4.1, more substantial changes would improve the quality of the code and reduce the effort required for proof.

[9] http://zeromq.org

3.1 Service Class Hierarchy

All services in UxAS inherit from a common abstract class named Service_Base that provides facilities for creating and configuring services. In particular, Service_Base creates a new service instance given only the name of the required service. This dynamic creation is necessary because UxAS instances are configured using service names listed in an XML configuration file and explains the use of pointers to designate dynamically allocated services.

Service_Base is a subclass of LMCP_Object_Network_Client_Base, the root abstract base class for all LMCP network-oriented client subclasses. This class provides the means for communicating LMCP messages over the network, and includes a thread, implemented as an Ada task, that actively sends and receives the messages.

Rather than defining Ada bindings to the C++ code, we implemented these classes in Ada because we want to be able to apply Ada features such as contracts and, in the future, extend the scope of the SPARK analysis to a larger portion of the code. To that end, although we follow the C++ design closely, we make changes for SPARK when necessary. For example, all state-changing functions in C++ are converted into procedures in Ada because SPARK does not allow functions to have side-effects. Similarly, we use bounded data types in place of unbounded types, e.g., String. We use a formally proven "dynamic bounded array" abstract data type for several of these replacements and use contracts extensively in the message serializer/deserializer class, requiring us to think through the intended usage scenarios and providing checks at run-time for our understanding.

3.2 Properties of Interest

Although a high-level description of the Automation Request Validator service exists on the OpenUxAS wiki[10], we found that there was insufficient detail there to identify meaningful properties. Instead, we examined the C++ code for the Automation Request Validator service to identify intent based on the current implementation. For the identification of intent, we restricted our focus to high-level understanding of the code and comments, rather than focusing on the details of the implementation.

The identified properties focus on the validation of specific, critical aspects of Automation Requests. A request contains several pieces of data: a list of entities, a list of operating regions, and a list of tasks. For a request to be valid, all these data should be checked to make sure that they have been previously declared and configured appropriately. This is done in a C++ function named isCheckAutomationRequestRequirements. This function takes an automation request and checks whether it is valid or not. Additionally, if the request is invalid, it computes and sends an error message to describe why the request was rejected. This function is translated in SPARK as a procedure (because a function cannot have side effects, including sending messages).

[10] https://github.com/afrl-rq/OpenUxAS/wiki/Core-Services-Description

To describe the functional behavior of isCheckAutomationRequestRequirements, we have introduced a SPARK function named Valid_Automation_Request that describes when an automation request should be valid. This function does not care about error messages; it simply describes validity in as concise a way as possible. Additionally, this function is specification-only, which means that it should not be used in the final executable. To make sure that this restriction is enforced, Valid_Automation_Request is annotated with the Ghost aspect (ghost code is removed by the compiler when assertion checking is disabled). To ensure that the definition stays in the specification part of the program, and is available for verification, we have defined Valid_Automation_Request directly as an expression function:

```
function Valid_Automation_Request
   (This    : Configuration_Data;
    Request : My_UniqueAutomationRequest) return Boolean
is
   -- Check entities
   (Check_For_Required_Entity_Configurations
   (...)

   -- Check operating regions
   and then Check_For_Required_Operating_Region_And_Keepin_Keepout_Zones
   (Operating_Region  => Get_OperatingRegion_From_OriginalRequest (Request),
      Operating_Regions => This.Available_Operating_Regions,
      KeepIn_Zones_Ids  => This.Available_KeepIn_Zones_Ids,
      KeepOut_Zones_Ids => This.Available_KeepOut_Zones_Ids)

   -- Check tasks
   and then Check_For_Required_Tasks_And_Task_Requirements
   (...)
with Ghost, Global => null;
```

The three subproperties are translated in the same way. For example, here is the function that checks the validity of operating regions:

```
function Check_For_Required_Operating_Region_And_Keepin_Keepout_Zones
   (Operating_Region   : Int64;
    Operating_Regions  : Operating_Region_Maps;
    KeepIn_Zones_Ids   : Int64_Set;
    KeepOut_Zones_Ids  : Int64_Set) return Boolean
is
   -- if there is an operating region, it should be listed in Operating_Regions
   (if Operating_Region ≠ 0 then Contains (Operating_Regions, Operating_Region)

    -- and all its associated keepin areas should be in KeepIn_Zones_Ids
    and then All_Elements_In
          (Element (Operating_Regions, Operating_Region).KeepInAreas,
           KeepIn_Zones_Ids)

    -- and all its associated keepout areas should be in KeepOut_Zones_Ids
    and then All_Elements_In
          (Element (Operating_Regions, Operating_Region).KeepOutAreas,
           KeepOut_Zones_Ids))
with Ghost;
```

That is, the requested operating region should have been previously stored in the operating-region map of the Automation Request Validator service, and its keep-in/keep-out areas should all be stored in the their respective sets.

3.3 Ada-SPARK Boundaries

The concrete Automation_Request_Validator_Service inherits from Service_Base, which inherits from LMCP_Object_Network_Client_Base. Both use constructs outside the SPARK subset, primarily pointers. Moreover, the Automation Request Validator service directly processes LMCP messages, which contain pointers and use container packages that are not amenable to formal analysis.

While changes are therefore required to enable analysis with SPARK, we avoid propagating these changes throughout the application and allow other parts of the application to use the full expressivity of Ada, in particular retaining the use of pointers. This approach promotes efficiency and stays as close as possible to the C++ code. Because the SPARK restrictions are mostly localized to the implementation of the Automation Request Validator service, this approach also simplifies modifying the integration between SPARK and Ada, for example if we change containers or take advantage of enhancements to SPARK.

The complexity of this approach lies in the interface between SPARK and Ada. When a SPARK function is called by Ada to validate a message, we must build a SPARK-compatible abstraction of the message. Rather than copying the message, we preserve the Ada types (including pointers and standard containers) and build abstractions on top of them so that they can be used in SPARK. These abstractions handle objects (e.g., messages, tasks, etc.) as black boxes and extract from them the required information in a SPARK-compatible way (e.g., translate standard containers to formal containers[11] or dereference pointers).

For example, the received Automation Request message is a pointer to an object of the Object inheritance class and is hidden from SPARK in a private type, with functions for converting to and from the pointer type and dereferencing:

```
package avtas.lmcp.object.SPARK_Boundary with SPARK_Mode is
   pragma Annotate (GNATprove, Terminating, SPARK_Boundary);

   type My_Object_Any is private;
   function Deref (X : My_Object_Any) return Object'Class with
      Global => null, Inline;
   function Wrap (X : Object_Any) return My_Object_Any with
      Global     => null, Inline,
      SPARK_Mode => Off;
   function Unwrap (X : My_Object_Any) return Object_Any with
      Global     => null, Inline,
      SPARK_Mode => Off;
private
   pragma SPARK_Mode (Off);
   type My_Object_Any is new Object_Any;
   (...)
end avtas.lmcp.object.SPARK_Boundary;
```

The SPARK code can dereference objects of type My_Object_Any using the Deref function. The functions to construct/destruct the abstractions (Wrap and Unwrap) are only accessible by the Ada code (they are marked SPARK_Mode ⇒Off).

[11] In addition to standard containers defined by Ada in the form of generic packages, SPARK includes a library of formal containers that have been designed specifically to facilitate proof.

4 Results

We developed a complete demonstration of the reimplemented, proven Automation Request Validator service, which is available on github[12]. We adapted an existing UxAS example that illustrates a UAV searching a waterway. The example includes a UxAS instance and the OpenAMASE simulator running as separate programs and communicating using ZeroMQ. Rather than integrate the Ada/SPARK into the C++ UxAS program, we run the service in a separate program. Our service receives messages from ZeroMQ and processes the Automation Request messages as if in the same UxAS instance as the C++ code.

Normally, a UxAS instance includes the Automation Request Validator service, which in this case would conflict with the Ada version since both would respond to Automation Request messages. Therefore, we disable the original Automation Request Validator service in OpenUxAS by removing it from the instance's XML configuration file. The C++ instance still receives Automation Request messages as they are injected but because none of its services process them, the intended UAV never begins the search. However, the XML file for the Ada version does include the Automation Request Validator service, so an instance is created, which validates Automation Request messages and then responds with Unique Automation Request messages. The UAV then performs the expected search.

4.1 Verification Results

Our goal was to verify both that the code fulfills its specification and that no errors can occur during its execution. We entirely achieved the first goal. We mostly achieved the second goal, with two notable exceptions.

First, we did not attempt to verify correct usage of the bounded strings and formal containers APIs. More precisely, we did not verify: the possible overflow of the error message string that is generated in response to an invalid request; the possible overflow of the data structures used to store messages and declared objects; or the uniqueness of keys in data structures, which requires reasoning about uniqueness of identifiers. These could be verified if we provided additional annotations and assumptions on inputs. We did not seek completeness because we believe these properties are insufficiently interesting to pollute other verification tasks with these concerns.

Second, the tool is unable to verify correctness of the part of the code in which, in C++, a classwide task object is cast to a specific task type depending on a string ID (its name). Ensuring the correctness of this code would require verifying globally the complete type hierarchy for all tasks, to make sure that each ID is never reused for a different task type. This problem may be seen as an incentive to refactor the code to use Ada membership tests instead of comparing string IDs, so that no such global invariant is required to ensure correctness.

[12] https://github.com/AdaCore/OpenUxAS, in the 'ada' branch

During our process of reverse engineering the specifications from the C++ code base and comments, we found only one error: a nested loop was used to find a match in two maps but both loops where iterating on the same map! We corrected this bug when we found it but demonstrated that it would have been detected by a formalization of the validity criteria.

Overall, the results we obtained using formal verification on the Automation Request Validator service are encouraging. However, the verification effort required to achieve this goal was significant because of two key challenges.

First, there was no appropriate high-level functional specification of what the service was supposed to do; we had to reverse engineer the specification from the C++ code and comments. Our specifications were validated by stakeholders.

Second, the code was not designed to be easily verified using SPARK; we had to abstract incompatible features, as detailed above. The abstractions could have been avoided by a global redesign of the code to more systematically use the formal containers and to eliminate pointers. Alternatively, the abstractions could have been avoided by improving the support in SPARK for excluded Ada features, such as pointers.

Because of these challenges, significant effort was required to define and express appropriate specifications in SPARK. Furthermore, actually verifying that the code conforms to its specification using the SPARK proof tool was challenging, because: (1) the code contains several loops, each requiring the use of a manually crafted loop invariant to act as a cut point for the tool; and (2) even with the code annotated and all the invariants supplied, we ran into provability issues. Indeed, the tool was overwhelmed by the amount of information it had to carry, mostly due to the number of different container instances employed. As a result, we had to manually guide the proof tool to complete the proofs by adding manual assertions in the code, sometimes at the expense of readability.

To help the provers, we primarily relied on two techniques.

First, we often restated the property we were trying to establish at several points in the program using `pragma Assert_And_Cut`. These pragmas not only check the property and add it to the context of subsequent checks like `pragma Assert` but also use the expression provided as a cut point. After the cut, the provers forget everything before the pragma and only remember the supplied property. For example, the code that checks that entities are properly configured is 175 lines long and includes 14 if statements and five loops, some of which are nested. At the conclusion of this code, we state that the `IsReady` flag really is the result of the expected computation using a `pragma Assert_And_Cut`:

```
pragma Assert_And_Cut
  (IsReady = Check_For_Required_Entity_Configurations
     (Entity_Ids      ⇒ EntityIds,
      Configurations  ⇒ This.Configs.Available_Configuration_Entity_Ids,
      States          ⇒ This.Configs.Available_State_Entity_Ids,
      Planning_States ⇒ Get_PlanningStates_Ids (Request)));
```

Thus we verify the property and help the verification of the remaining checks by forgetting the intermediate steps required by the computation up to that point.

Second, we introduced lemmas for often-reused reasoning. For example, the code sometimes performs computations that are hidden from the analysis, such

as sending messages to the outside world. While these computations modify the internal state of the service, they do not modify the configuration data such as the available entities. Unfortunately, the only mechanism provided by SPARK to state that a part of an object is unchanged by a subprogram call is Ada equality, which is fairly complex. Equality on an array, for instance, is the equality of elements: two arrays can be equal even if they have different bounds. As a result, proving that properties are preserved because two objects are equal can be nontrivial. For example, consider the contract of Send_Error_Response, which is used to send an error message if the request is invalid:

```
procedure Send_Error_Response
  (This              : in out Automation_Request_Validator_Service;
   Request           : My_UniqueAutomationRequest;
   ReasonForFailure  : Bounded_Dynamic_Strings.Sequence;
   ErrResponseID     : out Int64)
with Post => This.Configs'Old = This.Configs
and Same_Requests
  (Model (This.Requests_Waiting_For_Tasks),
   Model (This.Requests_Waiting_For_Tasks)'Old)
  and Same_Requests
    (Model (This.Pending_Requests),
     Model (This.Pending_Requests)'Old);
```

This contract states that both configuration data (This.Configs) and the request queues are left unchanged by the procedures.[13] When we call this procedure from our SPARK code, we would like to be able to deduce that if all requests were valid in the data configuration before the call, then they will be valid after the call. Unfortunately, this reasoning involves complex computations, as it relies on Ada equality for complicated data structures. Moreover, the validity of requests itself contains several (nested) quantified expressions. To help with these proofs, we introduced axioms in the form of ghost procedures with no effects; these axioms are used as lemmas in proofs. The premises are stated using preconditions; the conclusions are stated using postconditions. For example:

```
procedure Prove_Validity_Preserved
 (Data1, Data2 : Configuration_Data;
  R : My_UniqueAutomationRequest)
with
 Ghost,
 Global => null,
 Pre    => Data1 = Data2,
 Post   => Valid_Automation_Request (Data1, R) =
           Valid_Automation_Request (Data2, R);
```

This lemma states that if two configurations are equal, a request will have the same validity status in both. To use the lemma, we call it explicitly in the code:

```
declare
  ErrResponseID : Int64;
  Old_Confs : constant Configuration_Data := This.Configs with Ghost;
begin
  Send_Error_Response (This, Request, ReasonForFailure, ErrResponseID);
  Prove_Validity_Preserved (Old_Confs, This.Configs, Request);
end;
```

[13] We do not use Ada equality on the request queues: the requests contain parts which are hidden from SPARK, so SPARK does not know the meaning of equality for these queues; this is not the case, however, for the data configuration where we took care to only store SPARK-compatible information.

The use of both of these techniques made the proofs tractable *without* requiring a major redesign of the program. However, these techniques are costly in terms of lines of code. Check_Automation_Request_Requirements is about 200 lines of C++. Our verified version is approximately 410 lines long. Of these, the specification and contract is 20 lines, but depends on 130 lines of expression functions that help to express the property so that it is as readable as possible. The implementation contains roughly 45 lines of loop invariants and just over 100 lines of ghost code, including regular Assert pragmas, Assert_And_Cut pragmas, and calls to ghost lemmas with associated ghost state. The remainder — about 250 lines — is the Ada code, which we translated as closely as possible from the C++ version. In addition, we have some 50 lines of lemmas, most of which are automatically verified by the tool and do not require additional annotation.

5 Conclusion

We applied SPARK to OpenUxAS, AFRL's service-oriented architecture that provides core services supporting cooperative control and high-level command and control. In our application, we defined a partial specification for the Automation Request Validator service. We successfully proved that the implementation of the procedure intended to perform request validation satisfies the specification and additionally proved the absence of most run-time exceptions.

This work provides a foundation upon which we intend to build. In future work, we intend to extend the application of SPARK and Ada to additional services in UxAS and to investigate recently added support for ownership pointers to help simplify the application of SPARK. Ultimately, our goal is to provide a sufficient framework to enable us to formalize and prove interesting, application-relevant composition properties across the architecture.

References

1. Butler, R.W., Finelli, G.B.: The infeasibility of experimental quantification of life-critical software reliability. SIGSOFT Softw. Eng. Notes **16**(5), 66–76 (Sep 1991). https://doi.org/10.1145/123041.123054
2. Dross, C., Foliard, G., Jouanny, T., Matias, L., Matthews, S., Mota, J.M., Moy, Y., Pignard, P., Soulat, R.: Climbing the software assurance ladder-practical formal verification for reliable software (2018), https://www.adacore.com/uploads/techPapers/spark_avocs_2018.pdf
3. Kingston, D., Rasmussen, S., Humphrey, L.: Automated UAV tasks for search and surveillance. In: 2016 IEEE Conference on Control Applications (CCA). pp. 1–8 (Sep 2016). https://doi.org/10.1109/CCA.2016.7587813
4. Kingston, D., Beard, R.W., Holt, R.S.: Decentralized perimeter surveillance using a team of UAVs. IEEE Transactions on Robotics **24**(6), 1394–1404 (2008)
5. Rasmussen, S., Kingston, D., Humphrey, L.: A brief introduction to unmanned systems autonomy services (UxAS). pp. 257–268 (06 2018). https://doi.org/10.1109/ICUAS.2018.8453287