End-to-End Verification of Initial and Transition Properties of GR(1) Designs in SPARK*

Laura R. Humphrey[1], James Hamil[2], and Joffrey Huguet[3]

[1] Air Force Research Laboratory, WPAFB, OH 45433, USA
`laura.humphrey@us.af.mil`
[2] LinQuest Corp., Beavercreek, OH 45431, USA
`james.hamil.ctr@us.af.mil`
[3] AdaCore, F-75009 Paris, France
`huguet@adacore.com`

Abstract. Manually designing control logic for reactive systems is time-consuming and error-prone. An alternative is to automatically generate controllers using "correct-by-construction" synthesis approaches. Recently, there has been interest in synthesis from Generalized Reactivity(1) or GR(1) specifications, since the required computational complexity is relatively low, and several tools exist for synthesis from GR(1) specifications. However, while these tools implement synthesis approaches that are theoretically "correct-by-construction," errors in tool implementation can still lead to errors in synthesized controllers. We are therefore interested in "end-to-end" verification of synthesized controllers with respect to their original GR(1) specifications. Toward this end, we have modified Salty – a tool that produces executable software implementations of controllers from GR(1) specifications in a variety of programming languages – to produce implementations in SPARK. SPARK is both a programming language and associated set of verification tools, so it has the potential to enable the "end-to-end" verification we desire. In this paper, we discuss our experience to date using SPARK to implement controllers and verify them against a subset of properties comprising GR(1) specifications, namely system initial and system transition properties. We also discuss lessons learned about how to best encode controllers synthesized from GR(1) specifications in SPARK for verification, examples in which verification found unexpected controller behaviors, and caveats related to the interpretation of GR(1) specifications.

Keywords: Reactive synthesis · end-to-end verification · functional verification.

1 Introduction

Reactive systems must be capable of correctly responding to various inputs, e.g. originating from human users or events in the system's operational envi-

* Supported by AFRL contract FA8650-16-C-2642 and AFOSR grant RQCOR20-35. Distribution Statement A. Approved for public release: distribution unlimited. Case #88ABW-2020-0649.

ronment. The process of manually designing the control logic for such systems is both time-consuming and error prone. An alternative is to use "correct-by-construction" synthesis approaches to automatically generate a system's control logic directly from specifications, which can reduce both the amount of time needed for design and the likelihood of errors [1], [8], [10], [14]. In general, *synthesis* is the process of automatically generating a design from a specification. More specifically, *reactive synthesis* approaches generate designs in the context of an uncontrolled environment, assumptions about which are encoded in the specification. Reactive synthesis approaches tend to have high computational complexity, so there is particular interest in synthesis from Generalized Reactivity(1) or GR(1) specifications, the complexity of which is only polynomial in the size of the game graph encoded by the specification [3]. Synthesis from GR(1) specifications has been used to generate digital circuits [5] and controllers for teams of unmanned vehicles [2], ground robots [13], software-defined networks [16], and aircraft power distribution systems [18], to name a few.

GR(1) is a fragment of linear temporal logic. A GR(1) specification φ takes the form $\varphi = \varphi^e \rightarrow \varphi^s$, where φ^e encodes assumptions about the *environment* in which a system is to operate, and φ^s encodes guarantees the *system* should make under those assumptions [6]. More specifically, φ takes the form $\varphi = (\varphi^e_i \wedge \varphi^e_t \wedge \varphi^e_l) \rightarrow (\varphi^s_i \wedge \varphi^s_t \wedge \varphi^s_l)$, where φ^e_i and φ^s_i are *initial* properties, φ^e_t and φ^s_t are *transition* or safety properties, and φ^e_l and φ^s_l are *liveness* properties. For inputs in the set $\mathcal{I}$ controlled by the environment and outputs in the set $\mathcal{O}$ produced by the system, terms are defined as:

φ^e_i, φ^s_i -	Boolean formulas over $\mathcal{I}$ and $\mathcal{O}$, respectively, that characterize the initial state of the environment and system.
φ^e_t, φ^s_t -	Formulas of the form $\bigwedge_{j \in J} \Box B_j$, where each B_j is a Boolean combination of variables from $\mathcal{I} \cup \mathcal{O}$ and expressions of the form $\bigcirc v$, where $v \in \mathcal{I}$ for φ^e_t and $v \in \mathcal{I} \cup \mathcal{O}$ for φ^s_t. These encode properties that should always hold as well as rules for how inputs and outputs are allowed to change based on most recent input and output values.
φ^e_l, φ^s_l -	Formulas of the form $\bigwedge_{j \in J} \Box\Diamond B_j$, where each B_j is a Boolean formula over $\mathcal{I} \cup \mathcal{O}$. These encode properties that should hold infinitely often.

In this context, the temporal operators $\Box$ "always," $\Diamond$ "eventually," and $\bigcirc$ "next" have the following meanings. A formula of the form $\Box b$ holds if b is true at every time step, $\Diamond b$ holds if b is eventually true at some future time step, $\Box\Diamond b$ holds if b is true infinitely often in the future, and $\bigcirc b$ holds if b is true at the next time step. It is assumed that at each time step, the environment chooses an input from $\mathcal{I}$, then the system chooses an output from $\mathcal{O}$ in response. Synthesis from GR(1) specifications can therefore be viewed as a two-player game between the system and the environment, where the goal is for the system to satisfy φ^s as long as the environment satisfies φ^e. If this goal is achievable, then we say the specification is realizable, and the result is a *control protocol* or *strategy* for the system that can be expressed as a Moore machine.

Several tools for synthesis from GR(1) specifications are available. For example, RATSY [4] has a focus on circuit design and synthesizes designs encoded in BLIF, Verilog, and HIF. Similarly, Anzu [12] produces circuit designs in Verilog. LTLMoP [9] is focused on control of robots modeled as hybrid systems, and it synthesizes designs as hybrid controllers with handler modules to help connect controllers to simulated or real-world systems. TuLiP [17] has a similar focus and can synthesize controller implementations in Python. Slugs [6] is architected to allow users to tailor synthesis algorithms, e.g. to optimize criteria such as quick response, cost-optimality, and error-resilience, and it produces mathematical representations of controller designs. Salty [7] provides a front-end to Slugs that makes specifications easier to write and debug and a back-end that turns controller designs into executable software implementations in a variety of programming languages, including Python, C++, Java, and now SPARK. Though all of these tools implement synthesis algorithms that are theoretically "correct-by-construction," tool implementation errors could still result in errors in synthesized controllers. We are therefore interested in "end-to-end" verification of such controllers. Toward this end, we have extended Salty to produce software implementations of synthesized controller designs in SPARK.

SPARK, which is based on the Ada programming language, is both a programming language with a specification language and an associated verification toolset [11]. Though SPARK aims to perform fully automated verification, manual adaptation of the source code is in general necessary. For example, if one were to translate source code originally written in Ada to SPARK, one would have to remove unsupported features such as functions with side-effects, aliased variable names, exception handlers, etc. Formal verification of user-specified properties also requires manually writing specifications at the level of the source code in the form of contracts, e.g. preconditions and postconditions that summarize the assumptions and guarantees provided by individual subprograms.

There are different levels to which one can use SPARK to verify a program [15]. Generally, the level of verification performed during development is incremental, going from lowest to highest. Verification at the lowest level, colloquially referred to as *stone level*, is achieved when code is accepted by SPARK, since SPARK has stricter legality rules than Ada. Verification at the *bronze level* is achieved when flow analysis returns with no error; flow analysis checks for errors such as reads of uninitialized data or violations of user-specified data flow contracts. Verification at the *silver level* ensures that there will be no runtime errors when executing the program, e.g. division by zero or numeric overflow/underflow. Verification at the *gold level* consists of verifying key user specifications, such as type invariants and subprogram preconditions and postconditions; however, at this level, the specifications only partially specify the desired behavior of the code. Verification at the *platinum level* consists of verifying a complete set of specifications. At the moment, the SPARK code generated by Salty proves at gold level. In particular, SPARK is able to prove the system initial and system transition portions of the original GR(1) specification, where proof of system transition properties requires also proving a type invariant on the underlying Moore machine representation of the controller; this type invariant encodes that each state is reached by a specific set of input values and produces a specific set of output values. Platinum level would be reached if the liveness properties were expressed and proved, which we leave for future work.

In what follows, in Section 2 we describe the structure of synthesized controller implementations in SPARK by walking through a simple example involving a traffic light. To evaluate whether verification of synthesized SPARK controllers is feasible and has utility, we collected a corpus of examples from various sources. In Section 3, we take a detailed look at one of these examples, a controller that coordinates the actions of a team of unmanned air vehicles performing an escort mission, and we describe how SPARK revealed an error in the controller's specification. In Section 4, we give an overview of results for the rest of the examples, with a focus on scalability. In Section 5, we discuss lessons learned, including how to best structure synthesized controllers for proof. We end with concluding remarks in Section 6, including a discussion of future work.

2 Implementation and Verification of Synthesized Controllers in SPARK

We have extended the open source tool Salty[4] to produce software implementations of controllers synthesized from GR(1) specifications in SPARK. Given a GR(1) specification in Salty format, Salty does some preprocessing to sanity check, optimize, and translate the specification to Slugs format. Salty then calls Slugs to synthesize a controller design. Slugs provides the option of returning the controller design as a Moore machine expressed in text format, which is what Salty uses to create an executable software implementation of the controller. This Moore machine encodes all the states of the controller. It also encodes the transition relation between states, i.e. the controller's next state given the current state and the next set of input values. Every state then encodes a set of output values to be produced each time the controller transitions to that state. As in all Moore machines, each state produces exactly one set of output values. Controllers synthesized from GR(1) specifications have the additional property that for each state, there is a unique set of input values that brings the controller into that state, regardless of what the previous state was.

If SPARK is chosen as the target programming language, Salty translates this textual Moore machine representation of the controller to an implementation in SPARK. To give a brief overview, the Moore machine representing the controller is encoded as a record (analogous to a struct in C) that stores the controller's current state and a copy of the current input and output values, i.e. the values associated with the most recent transition. There is then a "move" procedure that implements the transition relation. This procedure changes the controller's state based on the next set of input values, produces the next set of output values, and updates its internal copy of current input and output values based on these most recent input and output values. Salty also synthesizes all of the annotations necessary to encode the specifications of interest and verify

[4] https://github.com/GaloisInc/salty/

that the controller satisfies those specifications. In terms of specifications, we seek to prove that the controller satisfies the system initial properties φ_i^s and system transition properties φ_t^s. Recall that GR(1) specifications take the form $(\varphi_i^e \wedge \varphi_t^e \wedge \varphi_l^e) \rightarrow (\varphi_i^s \wedge \varphi_t^s \wedge \varphi_l^s)$. Note however that for controllers synthesized from GR(1) specifications, system initial and transition properties should hold regardless of whether environment liveness properties hold. We are therefore interested in showing that $(\varphi_i^e \wedge \varphi_t^e) \rightarrow (\varphi_i^s \wedge \varphi_t^s)$. For the SPARK implementation, Salty therefore generates functions corresponding to φ_i^e and φ_t^e, which are used in the precondition for the move procedure. It also generates functions corresponding to φ_i^s and φ_t^s, which are used in the postcondition for the move procedure. If SPARK can verify that the move procedure satisfies the postcondition given the precondition, this verifies $(\varphi_i^e \wedge \varphi_t^e) \rightarrow (\varphi_i^s \wedge \varphi_t^s)$. In terms of additional annotations needed to prove this property, Salty generates "state to input mapping" and "state to output mapping" functions that encode which input and output values are associated with each state; these are used to define a type invariant on the controller. This is essentially all that is required to prove system initial and system transition properties of synthesized controllers in SPARK.

To understand synthesized SPARK controllers and annotations in more detail, consider a simple example involving a traffic light controller. Let the controller's single input be *tick*. Color changes occur in every state in which *tick* is true. The traffic light's color cycles infinitely over the sequence red $\rightarrow$ green $\rightarrow$ yellow $\rightarrow$ red $\rightarrow \ldots$. Let the output variables then be *red*, *yellow*, and *green*. The environment specifications are $\varphi_i^e = \top$, $\varphi_t^e = \top$, $\varphi_l^e = \square\Diamond tick$, i.e. *tick* can be true or false in the initial state and has no constraints on how it transitions from state to state, but it must be true infinitely often. The system initial and liveness specifications are $\varphi_i^s = red \wedge \neg yellow \wedge \neg green$ and $\varphi_l^s = \square\Diamond green$, i.e. the light starts as red but should infinitely often be green. The system transition specification is

$$\square(\bigcirc((red \wedge \neg yellow \wedge \neg green) \vee (\neg red \wedge yellow \wedge \neg green) \vee (\neg red \wedge \neg yellow \wedge green)) \wedge \quad (1)$$
$$(red \wedge \bigcirc tick \rightarrow \bigcirc green) \wedge (green \wedge \bigcirc tick \rightarrow \bigcirc yellow) \wedge (yellow \wedge \bigcirc tick \rightarrow \bigcirc red) \wedge \quad (2)$$
$$(red \wedge \bigcirc\neg tick \rightarrow red) \wedge (green \wedge \bigcirc\neg tick \rightarrow green) \wedge (yellow \wedge \bigcirc\neg tick \rightarrow yellow)) \quad (3)$$

That is, the light should only be one color at a time (1); the color should change from red $\rightarrow$ green $\rightarrow$ yellow $\rightarrow$ red $\rightarrow \ldots$ whenever *tick* is true (2); and the color should remain the same whenever *tick* is false (3).

In SPARK, subunits consist of a specification and a body. The package specification for this traffic light example is shown in Figure 1. Type `Controller` encodes a Moore machine representing the synthesized controller. In the public part of the specification, `Controller` (line 2) is declared as a private type so that the user cannot arbitrarily manipulate its state. In the private part of the specification, `Controller` (lines 33-37) is a record that stores the controller's internal state number, current input value(s), and current output value(s). The internal state number is always initialized to the last possible state number, i.e. the largest value for `State_Num`; this "controller initialization" state encodes the status of the controller before any inputs are received, and the only

```
1  package TrafficLight with SPARK_Mode is
2    type Controller is private;
3
4    type System is record
5      red: Boolean; yellow: Boolean; green: Boolean;
6    end record;
7
8    function Is_Init(C: Controller) return Boolean;
9    function Env_Init(tick: Boolean) return Boolean is (True);
10   function Sys_Init(S: System) return Boolean is
11     (S.red and not S.yellow and not S.green) with Ghost;
12
13   function Env_Trans(C: Controller; tick: Boolean) return Boolean
14     with Pre => (not Is_Init(C));
15   function Sys_Trans(C: Controller; tick: Boolean; S: System)
16     return Boolean with Pre => (not Is_Init(C)), Ghost;
17
18   procedure Move(C: in out Controller; tick: in Boolean; S: out System)
19   with
20     Pre => (if Is_Init(C) then Env_Init(tick) else Env_Trans(C, tick)),
21     Contract_Cases =>
22       (Is_Init(C) => Sys_Init(S) and (not Is_Init(C)),
23        others     => Sys_Trans(C'Old, tick, S) and (not Is_Init(C)));
24
25 private
26   function State_To_Input_Mapping(C: Controller) return Boolean
27     with Ghost;
28   function State_To_Output_Mapping(C: Controller) return Boolean
29     with Ghost;
30
31   subtype State_Num is Integer range 1 .. 7;
32
33   type Controller is record
34       State: State_Num := State_Num'Last; tick: Boolean; S: System;
35   end record
36     with Type_Invariant => (State_To_Input_Mapping(Controller) and
37                             State_To_Output_Mapping(Controller));
38
39 end TrafficLight;
```

Fig. 1. SPARK specification for a traffic light controller.

transition(s) out of this state are those allowed by φ_i^e. In this example, each input and output is of type `Boolean`. (Salty also allows for enumerations and integers, to be discussed later.) When there are multiple inputs or outputs, they are wrapped in a record of type `Environment` (not used here) or `System` (lines 4-6), respectively. There is only one input in this example, so it is not wrapped in a record, which is why there is no `Environment` record in Figure 1. Since `Controller` is a Moore machine, for each internal state, there is exactly one set of output values produced in that state. Furthermore, recall that controllers synthesized from GR(1) specifications have the additional property that there is exactly one set of input values that brings the controller into each state. `Controller` therefore includes a type invariant (lines 36-37) that captures this property, where functions `State_To_Input_Mapping` and `State_To_Output_Mapping` evaluate whether the controller's state corresponds to the expected input and output value(s), respectively. Both of these functions are declared (lines 26-29) with aspect `Ghost`, indicating that they are intended mainly for proof purposes, i.e. they will create verification conditions related to type `Controller` but will not be executed in the actual program, unless it is specified during compilation that they should be. Executing ghost code is often used for debugging, e.g. to check an unproved property through testing. In this case, removing the ghost code can save significant memory, since `State_To_Input_Mapping` and `State_To_Output_Mapping` internally encode large lookup tables that have to store information on every state of the controller, which is often thousands of states or even millions of states for the largest example in our database. The logic for these functions is given in the body (not shown), but for instance when `C.State = 1`, `State_To_Input_Mapping(C)` returns `True` if and only if `C.tick = False`, since *tick* being false brings the system into state 1; and `State_To_Output_Mapping(C)` returns `True` if and only if `C.S = System'(red => True, yellow => False, green => False)`, since *red* is true and *yellow* and *green* are false in state 1.

The public function `Is_Init` (line 8) checks whether the controller is in its initialization state, i.e. no inputs have yet been received. The public function `Env_Init` (line 9) checks whether input(s) satisfy φ_e^i. It is implemented as an expression function, i.e. the implementation is given directly in the specification, because all terms needed to define it are visible in the public part of the specification. In this example, since $\varphi_e^i = \top$ in the Salty specification, this automatically generated function always simply returns `True`. The public function `Sys_Init` (lines 10-11) checks whether outputs(s) satisfy φ_s^i. It is implemented as an expression function for the same reason. From the Salty specification, $\varphi_i^s = red \wedge \neg yellow \wedge \neg green$, so this function returns the value of the expression `S.red and not S.yellow and not S.green`. But unlike `Is_Init` and `Env_Init`, it is marked with aspect `Ghost` because it is mainly used for proof and does not need to be executed. `Is_Init` and `Env_Init` are used for proof but are also callable in functional code. We chose to make these functions non-ghost functions for reasons related to the meaning of GR(1) specifications. Recall that GR(1) specifications have the form $\varphi_e \rightarrow \varphi_s$. If φ_e ever becomes false, i.e. if the environment produces input value(s) that violate φ_e, then the specification as a whole is satisfied regardless of whether the system produces output value(s) that satisfy φ_s. In theory, the system could then produce arbitrary outputs and still satisfy the overall specification. In practice, we believe a user would generally want to know that the environment violated its specification, so that the user could either choose the system output value(s) explicitly or fall back to some other routine. Therefore, a user needs to be able to check inputs with `Env_Init` if `Is_Init` returns true, which is why both are callable. At the moment, they are not used in the code of the `Move` procedure. Public functions `Env_Trans` and `Sys_Trans` (lines 13-16) check whether the next set of input value(s) and output value(s) satisfy φ_e^t and φ_s^t, respectively. For the same reasons as above, `Sys_Trans` is a ghost function but `Env_Trans` is not. Note that `Env_Trans` has a precondition that the controller must not be in its initialization state, since φ_e^t can depend on both the current and next set of input value(s). This precondition is similarly necessary for `Sys_Trans`, since it can depend on both current and next input and output value(s). As with `Env_Init` and `Sys_Init`, the logic for these functions is synthesized from the Salty specification and implemented in the body (not shown), since they make use of input and output values stored in the `Controller`, whose fields are private.

The public procedure `Move` (lines 18-23) transitions the `Controller` based on its current internally stored values (i.e. state number and most recent input and output values) and next set of input value(s), and it produces the next set of output value(s). It has a precondition that if `Controller` is in its initialization state (i.e. it has not yet received any inputs), inputs must satisfy φ_e^i; otherwise they must satisfy φ_e^t. The aspect `Contract_Cases` specifies additional sets of preconditions paired with postconditions, where the set of all preconditions must be mutually exclusive and exhaustively cover the entire input space. The `others` keyword can be used to cover the set of all input conditions not covered in any explicit cases of the contract. Note that for the left-hand side of each case (i.e. left of `=>`), variable names refer to values before evaluation of the subprogram; for the right-hand side, they refer to values after evaluation. Therefore, the aspect `Old` can be used on the right to reference the value of a variable before evaluation of the subprogram. Combined with the previous precondition, `Contract_Cases` asserts that if the controller is in its initialization state, then after execution of `Move`, the first set of output value(s) produced should satisfy φ_s^i and the controller should no longer be in its initialization state. If the controller is not in its initialization state, then the output value(s) produced should satisfy φ_s^t, which is evaluated based on the most recent input and output values stored in `C'Old`, the next input value(s) just provided (in this case stored in `tick`), and the next output values just generated (in this case stored in the record `System`). This set of contract cases embodies our main proof goal, i.e. verification of system initial and transition properties from the original GR(1) specification.

A fragment of the body of `Move` is shown in Figure 2. Note that there are cases that can lead to `Program_Error`. This is because case statements require all possible cases to be covered, so we programmatically use `others` to cover all possible input combinations that would not be allowed due to φ_e^i or φ_e^t. In the traffic light example, these are unnecessary because all possible combinations of input values are allowed out of each state. In any case, SPARK will prove that such cases are not reachable if the preconditions of `Move` are met, i.e. if the environment satisfies its specification.

We briefly note that Salty includes language features that can result in different constructs being used to represent inputs and outputs in synthesized controllers, including enumerations and integers. For instance, enumerations encode that an enumerated input or output can have exactly one of a set of values at a time, as in the traffic light being exactly one color as expressed in φ_s^i and part (1) of φ_s^t. In such cases, enumerations or integers can make both specifications and code more compact and easier to read and understand. As a technical aside, during synthesis, enumerations and integers are translated to a bit vector representation along with additional initial and transition specifications that encode properties inherent to these types, such as values being mutually exclusive and

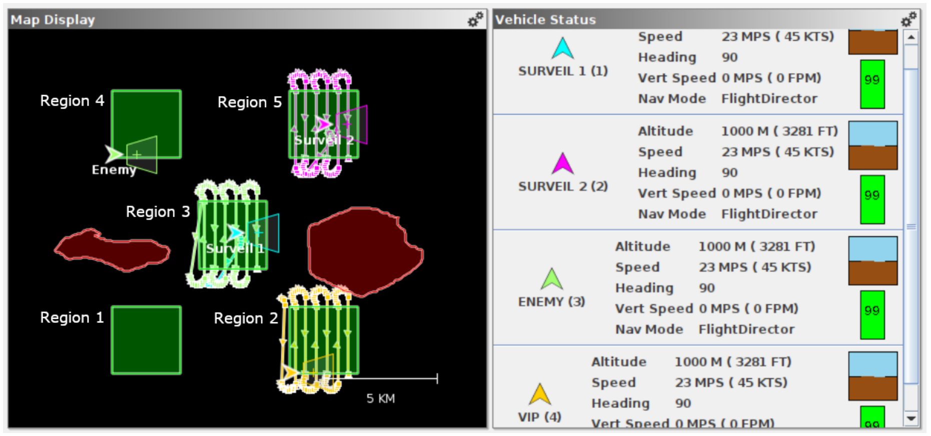
```
procedure Move(C: in out Controller; tick: in Boolean; S: out System) is
begin
  case C.State is
    when 1 =>
      case tick is
        when False =>
          C.State := 1;
          C.S.red := True; C.S.yellow := False; C.S.green := False;
        when True =>
          C.State := 3;
          C.S.red := False; C.S.yellow := False; C.S.green := True;
        when others =>
          raise Program_Error;
      end case;
        ...
     when 7 =>
      case tick is
        when False =>
          C.State := 1;
          C.S.red := True; C.S.yellow := False; C.S.green := False;
        when True =>
          C.State := 2;
          C.S.red := True; C.S.yellow := False; C.S.green := False;
        when others =>
          raise Program_Error;
      end case;
  end case;
  C.tick := tick; S := C.S;
end Move;
```

Fig. 2. The body of the Move procedure.

rules for addition and subtraction over integers. Once synthesis is complete, values are translated back to their original enumeration or integer representation.

3 Example Controller for a Team of UAVs

In order to evaluate the feasibility and utility of our approach, we collected GR(1) specifications from a variety of sources. This includes the Salty repository, which contains examples of GR(1) specifications for control of teams of unmanned air vehicles (UAVs) performing different missions. One of these encodes the rules for a “VIP Escort” mission, in which one UAV is designated as a “very important person” (VIP) that must always be “escorted” by a friendly surveillance UAV when it moves, and it must also be protected from an “enemy” UAV. The VIP and surveillance UAVs are controlled by the system, while the enemy UAV is controlled by the environment. The mission map contains regions that the UAVs can move between. “Escorting” consists of ensuring that 1) the VIP only enters regions previously visited by a surveillance UAV and 2) whenever the VIP changes regions, a surveillance UAV “tracks” it, i.e. moves with it between regions. “Protection” consists of ensuring that the VIP is never in the same region as the enemy UAV. Additional rules include constraints and liveness requirements on how the UAVs can move between regions and which regions they start in.



SURVEIL 1 (1)
Speed 23 MPS (45 KTS)
Heading 90
Vert Speed 0 MPS (0 FPM)
Nav Mode FlightDirector
99

SURVEIL 2 (2)
Altitude 1000 M (3281 FT)
Speed 23 MPS (45 KTS)
Heading 90
Vert Speed 0 MPS (0 FPM)
Nav Mode FlightDirector
99

ENEMY (3)
Altitude 1000 M (3281 FT)
Speed 23 MPS (45 KTS)
Heading 90
Vert Speed 0 MPS (0 FPM)
Nav Mode FlightDirector
99

VIP (4)
Altitude 1000 M (3281 FT)
Speed 23 MPS (45 KTS)
Heading 90
Vert Speed 0 MPS (0 FPM)

Fig. 3. A VIP escort multi-UAV mission as simulated in OpenAMASE using a Salty-synthesized controller that issues commands to the UAVs through OpenUxAS.

A particular instantiation of this mission is shown in Figure 3, as depicted in the open source Aerospace Multi-Agent Simulation Environment (OpenAMASE)[5]. It includes the VIP, two surveillance UAVs numbered 1 and 2, one enemy UAV, and five regions numbered 1 to 5. Environment inputs include integer variable $loc_e \in \{1 \ldots 5\}$, which encodes the current region of the enemy UAV, and Boolean variables sr_i for $i = \{1 \ldots 5\}$, where sr_i is true if and only if region i has been visited at some point by a surveillance UAV. System outputs include integer variables $loc_v, loc_{s1}, loc_{s2} \in \{1 \ldots 5\}$, which indicate the current region of the VIP and surveillance UAVs 1 and 2, and Boolean variables $vTrack_1$ and $vTrack_2$, which indicate whether surveillance UAVs 1 and 2 are executing a behavior to follow the VIP. Note that low-level control, e.g. waypoint planning and sensor steering, is implemented by the open source Unmanned Systems Autonomy Services (OpenUxAS)[6]. The controller synthesized by Salty implements high-level decision logic, and OpenUxAS monitors the state of the controller and translates its current output values to a set of UAV commands that are simulated in OpenAMASE. For example, when loc_v changes value to 2, OpenUxAS commands the VIP to follow a path from its current region to region 2, or when $vTrack_1$ changes from false to true, OpenUxAS commands surveillance UAV 1 to fly next to the VIP. The high-level controller synthesized by Salty makes some assumptions about the low-level behaviors implemented by OpenUxAS, in this case, mainly that all UAVs move at the same speed and transition to new regions at the same time. A workflow for connecting Salty-implemented controllers with OpenUxAS and OpenAmase is described in [7], and all of the scripts, configuration files, etc. needed to run this example are available on the Salty repository.

[5] https://github.com/afrl-rq/OpenAMASE
[6] https://github.com/afrl-rq/OpenUxAS

For this mission, the GR(1) specifications for the environment are:

$$\begin{aligned}
\varphi_e^i =\ & (loc_e = 4) \wedge \neg sr_1 \wedge \neg sr_2 \wedge sr_3 \wedge \neg sr_4 \wedge sr_5 && (4)\\
\varphi_e^t =\ & \bigwedge_{i=\{1\ldots5\}} \Box\Big((loc_{s1} = i) \vee (loc_{s2} = i) \rightarrow \bigcirc sr_i\Big) \wedge && (5)\\
& \bigwedge_{i=\{1\ldots5\}} \Box\Big(\big(\neg(loc_{s1} = i) \wedge \neg(loc_{s2} = i) \wedge \neg sr_i \rightarrow \bigcirc \neg sr_i\big) \wedge && (6)\\
& \qquad \big(sr_i \rightarrow \bigcirc sr_i\big)\Big) \wedge && (7)\\
& \Box\neg(loc_e = 1) \wedge \Box\neg(loc_e = 2) && (8)\\
\varphi_e^l =\ & \Box\Diamond\neg(loc_e = 3) \wedge \Box\Diamond\neg(loc_e = 4) \wedge \Box\Diamond\neg(loc_e = 5). && (9)
\end{aligned}$$

These express that (4) the enemy UAV starts in region 4, and regions 3 and 5 start as surveilled; (5) a region is considered to be surveilled after either one of the surveillance UAVs is in it; (6) a previously unsurveilled region remains unsurveilled if neither surveillance UAV is in it; (7) once a region is surveilled, it remains surveilled; (8) the enemy UAV cannot go to regions 1 or 2; and (9) the enemy UAV must infinitely often not be in each region 3, 4, and 5.

GR(1) specifications for the system are:

$$\begin{aligned}
\varphi_s^i =\ & (loc_v = 2) \wedge (loc_{s1} = 3) \wedge (loc_{s2} = 5) \wedge \neg vTrack_1 \wedge \neg vTrack_2 && (10)\\
\varphi_s^t =\ & \Box\Big(\neg(loc_v = \bigcirc loc_v) \rightarrow (\bigcirc vTrack_1 \vee \bigcirc vTrack_2)\Big) \wedge && (11)\\
& \bigwedge_{i=\{1\ldots2\}} \Box\Big(vTrack_i \rightarrow (sloc_i = loc_v)\Big) \wedge && (12)\\
& \bigwedge_{i=\{1\ldots5\}} \Box\Big((loc_v = i) \rightarrow \neg(loc_e = i)\Big) \wedge && (13)\\
& \bigwedge_{i=\{v,s1,s2\}} \Box\Big(\big(\bigcirc (loc_i = 1) \rightarrow (loc_i = 1) \vee (loc_i = 2) \vee (loc_i = 3)\big) \wedge && (14)\\
& \qquad \big(\bigcirc (loc_i = 2) \rightarrow (loc_i = 1) \vee (loc_i = 2) \vee (loc_i = 3)\big) \wedge && (15)\\
& \qquad \big(\bigcirc (loc_i = 3) \rightarrow \bigvee_{j=\{1\ldots5\}} (loc_i = j)\big) \wedge && (16)\\
& \qquad \big(\bigcirc (loc_i = 4) \rightarrow (loc_i = 3) \vee (loc_i = 4) \vee (loc_i = 5)\big) \wedge && (17)\\
& \qquad \big(\bigcirc (loc_i = 5) \rightarrow (loc_i = 3) \vee (loc_i = 4) \vee (loc_i = 5)\big)\Big) && (18)\\
\varphi_s^l =\ & \Box\Diamond(loc_v = 1) \wedge \Box\Diamond(loc_v = 5). && (19)
\end{aligned}$$

These express that (10) the VIP starts in region 2, surveillance UAV 1 in region 3, and surveillance UAV 2 in region 5, with neither surveillance UAV tracking the VIP; (11) the VIP does not change regions unless a surveillance UAV is tracking it; (12) a surveillance UAV can only track the VIP if they are in the same region at the same time; (13) the VIP cannot be in the same region as the enemy UAV at the same time; (14) and (15) the VIP and surveillance UAVs can move from regions 1 or 2 to regions 1, 2, or 3; (16) the VIP and surveillance UAVs can move from region 3 to any other region; (17) and (18) the VIP and surveillance UAVs can move from regions 4 or 5 to regions 3, 4, or 5; and (19) the VIP must go to regions 1 and 5 infinitely often.

We have chosen to describe this particular example in detail because during the process of extending Salty to produce SPARK implementations, we discovered a previously undetected problem with this example's specification. As written, the specification is realizable and produces what appears at a glance to be a reasonable controller with 97 states. In fact, we had previously run this example with OpenAMASE, OpenUxAS, and a Python controller synthesized by Salty. However, we did not originally notice in the Python implementation that 34 of the controller's 97 states do not have successors. Since no special logic was added to Salty to handle this situation, the generated SPARK code included empty case statement alternatives in the `Move` procedure. For example, in state 2 the case statement alternative is simply `when 2 =>`, with no statements in the body. This code failed to compile, since SPARK does not allow for fall-through behavior in case statements (nor does Ada); explicit statements are expected for each case statement alternative. In Python, this error went undetected; the transition relation for the controller is encoded as a map, and entries corresponding to states without successors simply had an empty list of "next state" values. We briefly note that if we had encoded the controller using a map in SPARK, the error would still have been detected through SPARK analysis rather than a syntactic check of the code; the case statement encoding was chosen for efficiency reasons to be discussed in Section 5.

This error is the result of a subtlety of the semantics of GR(1) specifications. Recall that GR(1) specifications take the form $\varphi_e \rightarrow \varphi_s$. Obviously a specification of this form is satisfied if φ_e and φ_s are both true, but it is also satisfied if φ_e is false. Also recall from the introduction that GR(1) specifications are interpreted in the context of a two-player game in which the environment takes its turn first and the system takes its turn second. The issue here is that the environment is able to take transitions that will necessarily cause it to violate φ_e^t in the next time step. Note that φ_e^t contains terms of the form $\Box\neg p$, specifically $\Box\neg(loc_e = i)$ for $i = \{1, 2\}$ (8). Note that a term of the form $\Box\neg p$ is not the same as a term of the form $\Box \bigcirc \neg p$. The distinction is important. If the environment chooses p for the "next" time step, this does not violate $\Box\neg p$ in the "current" time step. However, once the next state is reached, p becomes the new "current" value, and $\Box\neg p$ will now be violated no matter what the environment chooses. Generally, specifications should follow the latter form $\Box \bigcirc \neg p$, which prohibits the environment from choosing p in the "next" time step. This was indeed an error in these specifications, so we changed $\Box\neg(loc_e = 1) \wedge \Box\neg(loc_e = 2)$ to $\Box \bigcirc \neg(loc_e = 1) \wedge \Box \bigcirc \neg(loc_e = 2)$ in (8). However, we also modified Salty to `raise Program_Error` in cases with no successors and checked that we were still able to prove $\varphi_e \rightarrow \varphi_s$ (since reaching these cases would require violating the precondition φ_e). Such cases amount to instances in which the precondition on `Move` would have to be violated, which is why we allow the user to execute `Env_Trans` as discussed in Section 2. We also plan to have Salty issue a warning when there are states with no successors, since such cases are likely unintended.

4 Results

To further evaluate the utility and feasibility of our approach, we pulled additional example GR(1) specifications from a variety of sources including Anzu, LTLMoP, TuLiP, Slugs, and Salty, all of which make their examples publicly available for download. GR(1) specifications in Salty format, synthesized SPARK packages, and SPARK analysis results for all of these examples are available on the Salty GitHub repository, including the traffic light example of the previous section. We note that while some examples are small and simple, e.g. demos along the lines of the traffic light example, there are many in our collection that are more realistic. For instance, Anzu has controller specifications for a generalized IBM buffer and an AMBA bus. LTLMoP and TuLiP have specifications for robot controllers that have been demonstrated on simulated and/or real robots. And Salty has specifications for controllers to coordinate the actions of teams of vehicles that have been demonstrated in simulation.

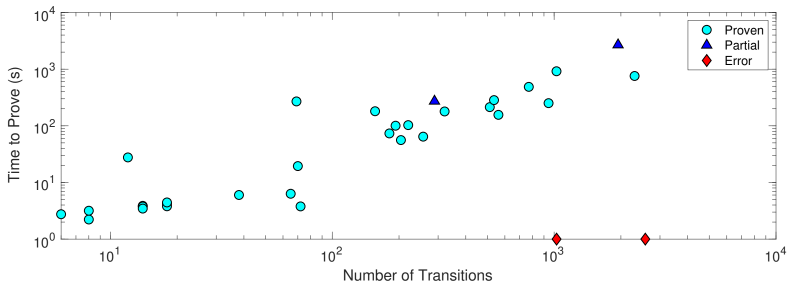


Fig. 4. Timing results for example SPARK controllers as a function of number of transitions in the controller. "Proven" examples were fully verified, "partial" examples were partially verified, and "error" examples were too big to analyze.

Figure 4 shows the amount of time needed to analyze examples as a function of total number of transitions in the Moore machine representing the controller, with examples that could not be analyzed due to memory errors set to 1. Results were generated on a Linux VM given 24GB RAM and 4 processors on a MacBook Pro with a 2.9 GHz Intel Core i9 with 32GB RAM. We ran 40 examples in total. Results for 33 examples are plotted. On most examples with less than 4000 transitions, SPARK was able to completely verify/prove the synthesized code complies with its specification. Examples that had more than 4000 transitions (the 7 unplotted examples) required too much memory to analyze, resulting in errors when attempting to verify them in SPARK.

Of examples with less than 4000 transitions, two resulted in errors. These two examples had abnormally large specifications consisting of approximately 1000 atomic propositions each, whereas most other examples with a similar number of transitions had 500 or less. Such cases occur, e.g. when systems include a large number of inputs and/or outputs but have very tight specifications on how they can transition, leading to large specifications and therefore lengthy pre- and postconditions but relatively small controllers.

Two examples with less than 4000 transitions could only be partially proven. One was relatively large, with about 2000 transitions. The other had arithmetic terms in its specification (Salty and Slugs support integer inputs and outputs); we are investigating why this example does not fully prove, since we feel that SPARK should be capable of fully proving this example.

5 Lessons Learned

Throughout the process of synthesizing and attempting to verify controllers in SPARK, we learned several lessons, both about SPARK and about some of the finer points of GR(1) specifications.

In terms of encoding SPARK controllers for verification, we originally tried to mirror the approach taken in other Salty language targets by building a static lookup table for state transitions. To do this, we tried to create an array of `Formal_Hashed_Maps` indexed by `State_Num`, where keys were derived from environment input values and used to look up the next `State_Num` and corresponding system output values. Ghost functions consisting of nested quantified expressions were used to check that in each state, specification properties held using input and output values encoded by the current state and all states reachable as contained in the hashed maps. These functions comprised the postcondition of a function that initialized the controller's lookup table. The body of the `Move` procedure simply retrieved the outputs and next state from the lookup table using its stored `State_Num` and `Environment` input. The public portion of the SPARK specification was largely unchanged. This approach was only able to prove the smallest of examples in a reasonable amount of time.

While the use of formal containers was intuitive, they are more complex to reason about in terms of proof because they require reasoning about models of the containers. Encoding the lookup table as a case statement is more straightforward, because for instance, it is "obvious" to the underlying solvers that state transitions are static and that a transition exists for every possible input, since case statements must be exhaustive. Encoding the `Move` procedure as a case statement still has some issues, mainly that (1) the generated code can be quite long, leading to memory errors when trying to prove the subprogram with SPARK, and (2) since the solvers prove all case statement alternatives at the same time and the number of case statement alternatives grows exponentially with the number of inputs, sometimes the solvers are not able to prove the postcondition. A solution to both problems could be to split the `Move` procedure into several smaller procedures. This would allow SPARK to apply modular analysis on several smaller procedures, thus enabling the proof on larger files. We are currently investigating ways to split up the procedure that does not accidentally create more difficulties for the underlying solvers.

The process of encoding and analyzing controllers in SPARK did reveal some unexpected behaviors. First, as discussed in Section 3, there were two example controllers[7] with specifications that resulted in states with no successors. As a result, these controllers contained empty case statements in the `Move` procedure. We had previously tested the Salty-synthesized controller in Python for the example in Section 3 and had not noticed the error, though it would have resulted in an unhandled runtime exception if one of the states without successors had been reached in the Python implementation. Second, a meta-analysis of SPARK timing results also revealed that other examples in our database did not have any inputs, i.e. they amounted to synthesizing a system independent of an environment. In those cases, we had specifications for a non-existent environment that were vacuous, and this caused SPARK to take an abnormally long amount of time to verify these controllers, given their relatively small size. These controllers did not have errors per se, but they were inefficiently encoded. We plan to modify Salty to handle such cases by removing the environment, functions over the environment, and all references to the environment in all pre- and post-conditions. This greatly decreases verification time and also reduces the size and increases the efficiency of the code.

6 Conclusions

We were able to successfully use SPARK to verify safety and transition properties of moderately sized controllers synthesized by Salty from GR(1) specifications. Encoding the controllers and all of the annotations necessary for these controllers to prove automatically was relatively straighforward, and it was satisfying to be able to generate proofs using a single tool rather than having to use multiple tools to perform verification. Furthermore, the act of performing “end-to-end” verification with SPARK on such controllers was valuable because (1) it revealed a type of specification error in some examples that would result in runtime errors in other Salty target languages, and (2) it revealed cases in which controllers were inefficiently encoded, i.e. when there is no environment.

In terms of future work, we can potentially improve the scalability of our approach by decomposing the `Move` procedure into several subprocedures, as discussed in the previous section. We are also interested in expressing and proving liveness properties. Liveness properties will be more challenging to verify because they necessarily require reasoning about future states beyond the “next” state. Verifying system liveness in SPARK will require something like encoding a lookahead buffer and showing that certain states will inevitably be reached when the environment satisfies its specification, which can itself also include liveness terms. This is likely to result in complex first-order formulas with alternating quantification over time, which are notoriously hard to handle in automated solvers, so discharging the resulting proof obligations may prove to be a challenge. To tackle this issue, collaboration with a model checker performing verification at the level of the input language might be more appropriate.

[7] Salty’s vip_orig.salt and Anzu’s arbiter.salt

References

1. Alur, R., Moarref, S., Topcu, U.: Compositional synthesis of reactive controllers for multi-agent systems. In: Int. Conf. Computer Aided Verification (CAV). pp. 251–269. Springer (2016)
2. Apker, T.B., Johnson, B., Humphrey, L.R.: LTL templates for play-calling supervisory control. In: AIAA Infotech@Aerospace. AIAA (2016)
3. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Yaniv, S.: Synthesis of Reactive(1) designs. J. Computer and System Sciences **78**(3), 911–938 (2012)
4. Bloem, R., Cimatti, A., Greimel, K., Hofferek, G., Könighofer, R., Roveri, M., Schuppan, V., Seeber, R.: RATSY–a new requirements analysis tool with synthesis. In: Int. Conf. Computer Aided Verification (CAV). Springer (2010)
5. Ehlers, R., Könighofer, R., Hofferek, G.: Symbolically synthesizing small circuits. In: IEEE Formal Methods in Computer-Aided Design (FMCAD). pp. 91–100. IEEE (2012)
6. Ehlers, R., Raman, V.: Slugs: Extensible GR(1) synthesis. In: Int. Conf. Computer Aided Verification (CAV), pp. 333–339. Springer (2016)
7. Elliott, T., Alshiekh, M., Humphrey, L.R., Pike, L., Topcu, U.: Salty–a domain specific language for GR(1) specifications and designs. In: 2019 Int. Conf. Robotics and Automation (ICRA). pp. 4545–4551. IEEE (2019)
8. Fainekos, G.E., Girard, A., Kress-Gazit, H., Pappas, G.J.: Temporal logic motion planning for dynamic robots. Automatica **45**(2), 343–352 (2009)
9. Finucane, C., Jing, G., Kress-Gazit, H.: LTLMoP: Experimenting with language, temporal logic and robot control. In: IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS). pp. 1988–1993. IEEE (2010)
10. Guo, M., Tumova, J., Dimarogonas, D.V.: Cooperative decentralized multi-agent control under local LTL tasks and connectivity constraints. In: IEEE Conf. Decision and Control (CDC). pp. 75–80. IEEE (2014)
11. Hoang, D., Moy, Y., Wallenburg, A., Chapman, R.: SPARK 2014 and GNATprove. Int. J. Software Tools for Technology Transfer **17**(6), 695–707 (2015)
12. Jobstmann, B., Galler, S., Weiglhofer, M., Bloem, R.: Anzu: A tool for property synthesis. In: Int. Conf. Computer Aided Verification (CAV). pp. 258–262. Springer (2007)
13. Kress-Gazit, H., Fainekos, G.E., Pappas, G.J.: Where's Waldo? Sensor-based temporal logic motion planning. In: IEEE Int. Conf. Robotics and Automation (ICRA). pp. 3116–3121. IEEE (2007)
14. Kupermann, O., Vardi, M.: Synthesizing distributed systems. In: IEEE Symp. Logic in Computer Science. pp. 389–398. IEEE (2001)
15. Moy, Y.: Climbing the software assurance ladder-practical formal verification for reliable software. Electronic Communications of the EASST **76** (2019)
16. Wang, A., Moarref, S., Loo, B.T., Topcu, U., Scedrov, A.: Automated synthesis of reactive controllers for software-defined networks. In: IEEE Int. Conf. Network Protocols (ICNP). pp. 1–6. IEEE (2013)
17. Wongpiromsarn, T., Topcu, U., Ozay, N., Xu, H., Murray, R.M.: TuLiP: A software toolbox for receding horizon temporal logic planning. In: Int. Conf. Hybrid Systems: Computation and Control. pp. 313–314. HSCC '11, ACM (2011)
18. Xu, H., Topcu, U., Murray, R.M.: A case study on reactive protocols for aircraft electric power distribution. In: IEEE Conf. Decision and Control (CDC). pp. 1124–1129. IEEE (2012)