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

Laura Humphrey, James Hamil

AFRL, Aerospace Systems Directorate, Autonomous Controls Branch (AFRL/RQQA)

Joffrey Huguet

AdaCore

September 2020

Outline

- GR(1) specifications & synthesis
- “End-to-end” verification in this context
- SPARK for end-to-end verification of synthesized GR(1) designs
- Case study: Multi-vehicle controller
- Results on a corpus of GR(1) specifications
- Summary: lessons learned & future work

GR(1) Specifications & Synthesis

GR(1) Specifications

- GR(1) or Generalized Reactivity(1) specifications
  - Describe reactive systems that respond to an environment
  - Subset of linear temporal logic
  - General form: $\varphi_e \rightarrow \varphi_s$

- Interpreted as two-player game between system & environment
  - Environment e goes first, controls "input" variables from set $\mathcal{I}$
  - System s goes second, controls "output" variables from set $\mathcal{O}$

- Are "realizable" if system strategy/design exists such that
  - System can satisfy φ_s after each step environment satisfies φ_e
  - Environment is forced to violate φ_e

- Result is "controller" with states S encoding strategy/design $c : S \times \Sigma_{\mathcal{I}} \mapsto S \times \Sigma_{\mathcal{O}}$

GR(1) Specifications as Properties

φ^e & φ^s each broken into initial, transition, & liveness property terms

$$\varphi_i^e \wedge \varphi_t^e \wedge \varphi_l^e \rightarrow \varphi_i^s \wedge \varphi_t^s \wedge \varphi_l^s$$

Temporal operators

$\bigcirc$ – "next" $\square$ – "always" $\square\lozenge$ – "always eventually"

Property term definitions

φ_i^e, φ_i^s - Initial : Boolean formulas over $\mathcal{I}$ & $\mathcal{O}$, respectively

φ_t^e, φ_t^s - Transition : $\bigwedge_{j \in J} \square B_j$, where each B_j is a Boolean formula over
- Variables from $\mathcal{I} \cup \mathcal{O}$
- Expressions $\bigcirc v$, where $v \in \mathcal{I}$ for φ_t^e & $v \in \mathcal{I} \cup \mathcal{O}$ for φ_t^s

φ_l^e, φ_l^s - Liveness : $\bigwedge_{j \in J} \square\lozenge B_j$, where each B_j is a Boolean formula over $\mathcal{I} \cup \mathcal{O}$

GR(1) Example

- Consider a traffic light
  - Input: “tick”
  - Outputs: “red,” “yellow,” “green”
  - Output changes when “tick” is true, stays the same when “tick” is false
  - Ex. 1: red, red, red, …
  - Ex. 2: red, yellow, green, red, …
  - Ex. 3: red, red, yellow, yellow, green, green, …
- Environment properties

$$\varphi_i^e = \top \qquad \varphi_t^e = \top \qquad \varphi_l^e = \Box\Diamond tick$$

- System initial & liveness properties

$$\varphi_i^s = red \wedge \neg yellow \wedge \neg green$$

$$\varphi_l^s = \Box\Diamond green$$

- System transition properties

<u>Mutual exclusion</u>

$$\varphi_s^t = \Box \bigcirc ((red \wedge \neg yellow \wedge \neg green) \vee (\neg red \wedge yellow \wedge \neg green) \vee (\neg red \wedge \neg yellow \wedge green)) \wedge$$

<u>Change on tick</u>

$$\Box ((red \wedge \bigcirc tick \rightarrow \bigcirc green) \wedge (green \wedge \bigcirc tick \rightarrow \bigcirc yellow) \wedge (yellow \wedge \bigcirc tick \rightarrow \bigcirc red)) \wedge$$

<u>No change without tick</u>

$$\Box ((red \wedge \bigcirc \neg tick \rightarrow \bigcirc red) \wedge (green \wedge \bigcirc \neg tick \rightarrow \bigcirc green) \wedge (yellow \wedge \bigcirc \neg tick \rightarrow \bigcirc yellow))$$

GR(1) & Synthesis

- Given a GR(1) specification, we can
  - Verify design satisfies it
  - Efficiently synthesize design directly from it
- Ref. [1] contains proof synthesis procedure is "correct-by-construction"
- Synthesis tools exist for different domains[2-6]: robotic systems, hybrid systems, multi-agent systems, digital circuits, etc.
- Only a few generate software implementations of synthesized designs

End-to-End Verification

GR(1) Synthesis & End-to-End Verification

- Question: If synthesis from GR(1) specifications is “correct-by-construction,” why care about end-to-end verification?

- Answer: Three possible sources of errors
  - Theoretical “proof” of the synthesis procedure[7-8]
  - Implementation of the synthesis procedure
  - Translation of a synthesized design to a software implementation

GR(1) Synthesis & End-to-End Verification

- Question: If synthesis from GR(1) specifications is "correct-by-construction," why care about end-to-end verification?

- Answer: Three possible sources of errors
  - Theoretical "proof" of the synthesis procedure[7-8]
  - Implementation of the synthesis procedure
  - Translation of a synthesized design to a software implementation

- Additional answer: specification errors

SPARK for End-to-End Verification of GR(1) Designs

SPARK for End-to-End Verification of GR(1) Designs

- SPARK[9-10] is
  - Programming language based on Ada
  - Associated set of auto-active verification tools
- Our goal
  - Translate synthesized GR(1) designs to SPARK
  - Verify SPARK implementations satisfy original specifications
- Our broad approach
  - Modify tool Salty[6] to generate SPARK implementations & annotations needed for SPARK to automatically verify implementations against specifications
  - Perform synthesis & verification on examples from many sources[2-6]
- Limitations
  - Currently only address initial & transition properties: $\varphi_i^e \wedge \varphi_t^e \rightarrow \varphi_i^s \wedge \varphi_t^s$

SPARK Implementation of GR(1) Design: Summary

- Multiple inputs wrapped in `Environment` record
- Multiple outputs wrapped in `System` record
- Synthesized design implemented as type `Controller`
  - Has field to store state
  - Has fields to store most recent input & output values (for evaluating transition properties with "next" operator)
- Type `Controller` has `Move` procedure that takes inputs & produces outputs
- Functions `Env_Init`, `Env_Trans`, `Sys_Init`, `Sys_Trans` to evaluate $\varphi_i^e, \varphi_t^e, \varphi_i^s, \varphi_t^s$
- Function `Is_Init` to check whether `Controller` is in initial state

SPARK Implementation of GR(1) Design: Proof Notes

- Preconditions & postconditions on `Move` encode $\varphi_i^e \wedge \varphi_t^e \rightarrow \varphi_i^s \wedge \varphi_t^s$

- In GR(1) designs, each state corresponds to unique set of input & output values
  - Define `Type_Invariant` for `Controller` over Boolean functions `State_To_Input_Mapping` & `State_To_Output_Mapping`
  - This is the only thing SPARK needed to automatically prove `Move` postconditions

- "Ghost" code versus executable code
  - `Is_Init`, `Env_Init`, `Env_Trans` executable so user can monitor for violations of φ_e
  - `Sys_Init`, `Sys_Trans`, `State_To_Input_Mapping`, `State_To_Output_Mapping` are "ghost" code mainly used for proof

SPARK Implementation of GR(1) Design: Traffic Light

- Let’s briefly walk through parts of synthesized traffic light design in SPARK
- Omit Salty-formatted specifications & stick with mathematical representation
- Briefly note Salty has features that make GR(1) specifications easier to write
  - Use of enumeration & integer types for inputs & outputs
  - Arithmetic operators in specifications with integer types
  - User-defined macros
- Enumeration & integer types retained in synthesized SPARK implementations

SPARK package specification for the synthesized traffic light design

```
package TrafficLight with SPARK_Mode is
  type Controller is private;

  type System is record
    red: Boolean; yellow: Boolean; green: Boolean;
  end record;

  function Is_Init(C: Controller) return Boolean;
  function Env_Init(tick: Boolean) return Boolean is (True);
  function Sys_Init(S: System)     return Boolean is (S.red and not S.yellow and not S.green) with Ghost;

  function Env_Trans(C: Controller; tick: Boolean) return Boolean
    with Pre => (not Is_Init(C));
  function Sys_Trans(C: Controller; tick: Boolean; S: System) return Boolean
    with Pre => (not Is_Init(C)), Ghost;

  procedure Move(C: in out Controller; tick: in Boolean; S: out System)
  with Pre =>             (if Is_Init(C) then Env_Init(tick) else Env_Trans(C, tick)),
       Contract_Cases => (Is_Init(C) => Sys_Init(S) and (not Is_Init(C)),
                          others     => Sys_Trans(C'Old, tick, S) and (not Is_Init(C)));
private
  subtype State_Num is Integer range 1 .. 7;
  function State_To_Input_Mapping(C: Controller) return Boolean with Ghost;
  function State_To_Output_Mapping(C: Controller) return Boolean with Ghost;

  type Controller is record
      State: State_Num := State_Num'Last; tick: Boolean; S: System;
  end record;
    with Type_Invariant => (State_To_Input_Mapping(Controller) and State_To_Output_Mapping(Controller));
end TrafficLight;
```

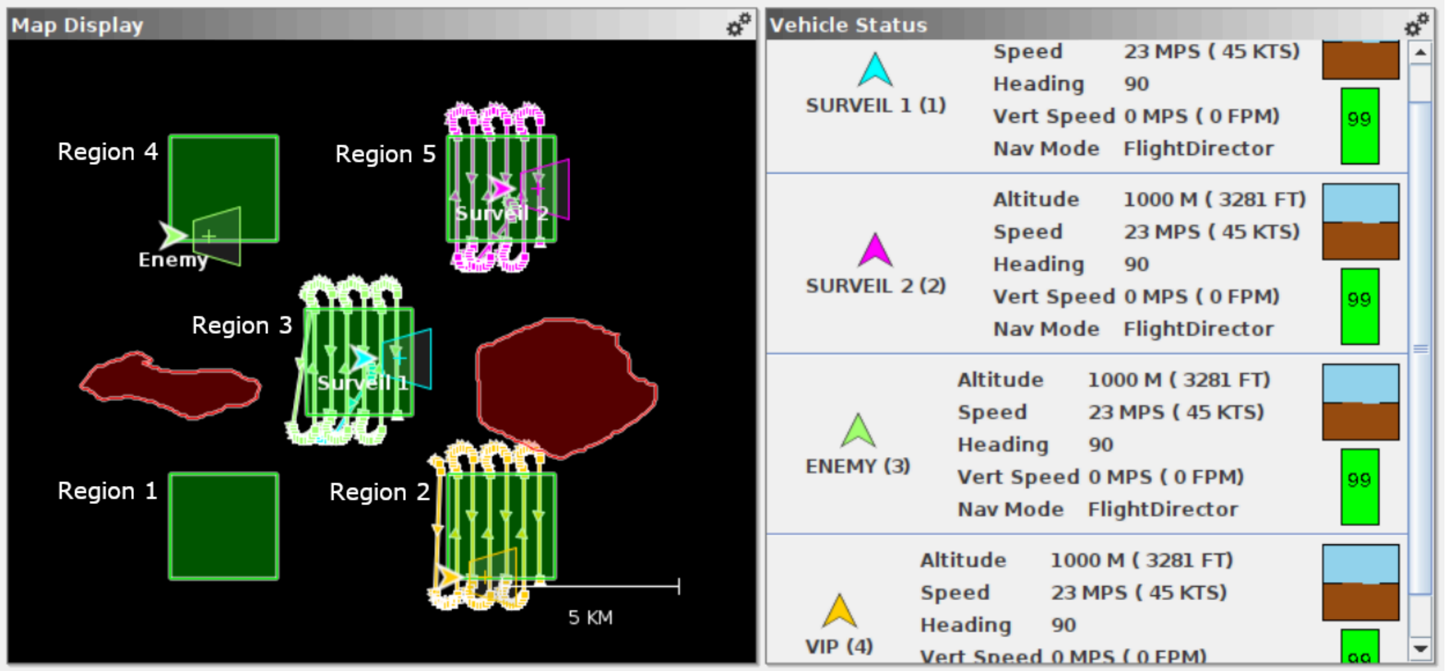
SPARK body for the `Move` procedure for the synthesized traffic light design

```
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;
```

Case Study:
Multi-Vehicle Controller

Case Study: Multi-Vehicle Controller

- Team of "friendly"/system unmanned air vehicles (UAVs) evading "enemy"/environment UAV
  - One friendly UAV is a "Very Important Person" or VIP
  - Two friendly "surveillance" UAVs must "protect" & "escort" the VIP
- Map is divided into five regions
- "Protection": VIP is never in the same region as the enemy
- "Escorting": VIP can only move
  - Into region previously visited/ surveilled by a surveillance UAV
  - Surveillance UAV moves with it
- Certain regions only reachable from Region 3
- Enemy UAV must infinitely often leave certain regions
- VIP must infinitely often move to certain regions



Specification Inputs & Outputs

Inputs	
loc_e	- enemy location
sr_i	- location i surveilled

Outputs	
loc_v	- VIP location
loc_{si}	- Surveillance UAV i location
$vTrack_i$	- Surveillance UAV i is tracking VIP

Environment Specifications

$$\varphi_e^i = (loc_e = 4) \wedge \neg sr_1 \wedge \neg sr_2 \wedge sr_3 \wedge \neg sr_4 \wedge sr_5$$

$$\varphi_e^t = \bigwedge_{i=\{1\ldots5\}} \Box\Big((loc_{s1} = i) \vee (loc_{s2} = i) \rightarrow \bigcirc sr_i\Big) \wedge$$

$$\bigwedge_{i=\{1\ldots5\}} \Box\Big((\neg(loc_{s1} = i) \wedge \neg(loc_{s2} = i) \wedge \neg sr_i \rightarrow \bigcirc \neg sr_i) \wedge$$

$$(sr_i \rightarrow \bigcirc sr_i)\Big) \wedge$$

$$\Box\neg(loc_e = 1) \wedge \Box\neg(loc_e = 2)$$

$$\varphi_e^l = \Box\Diamond\neg(loc_e = 3) \wedge \Box\Diamond\neg(loc_e = 4) \wedge \Box\Diamond\neg(loc_e = 5)$$

Inputs		
loc_e	-	enemy location
sr_i	-	location i surveilled

Outputs		
loc_v	-	VIP location
loc_{si}	-	Surveillance UAV i location
$vTrack_i$	-	Surveillance UAV i is tracking VIP

System Specifications

$$\varphi_s^i = (loc_v = 2) \wedge (loc_{s1} = 3) \wedge (loc_{s2} = 5) \wedge \neg vTrack_1 \wedge \neg vTrack_2$$

$$\varphi_s^t = \Box\Big(\neg(loc_v = \bigcirc loc_v) \rightarrow (\bigcirc vTrack_1 \vee \bigcirc vTrack_2)\Big) \wedge$$

$$\bigwedge_{i=\{1\ldots2\}} \Box\Big(vTrack_i \rightarrow (sloc_i = loc_v)\Big) \wedge$$

$$\bigwedge_{i=\{1\ldots5\}} \Box\Big((loc_v = i) \rightarrow \neg(loc_e = i)\Big) \wedge$$

$$\bigwedge_{i=\{v,s1,s2\}} \Box\Big((\bigcirc (loc_i = 1) \rightarrow (loc_i = 1) \vee (loc_i = 2) \vee (loc_i = 3)) \wedge$$

$$(\bigcirc (loc_i = 2) \rightarrow (loc_i = 1) \vee (loc_i = 2) \vee (loc_i = 3)) \wedge$$

$$(\bigcirc (loc_i = 3) \rightarrow \bigvee_{j=\{1\ldots5\}} (loc_i = j)) \wedge$$

$$(\bigcirc (loc_i = 4) \rightarrow (loc_i = 3) \vee (loc_i = 4) \vee (loc_i = 5)) \wedge$$

$$(\bigcirc (loc_i = 5) \rightarrow (loc_i = 3) \vee (loc_i = 4) \vee (loc_i = 5))\Big)$$

$$\varphi_s^l = \Box\Diamond(loc_v = 1) \wedge \Box\Diamond(loc_v = 5)$$

Inputs		
loc_e	-	enemy location
sr_i	-	location i surveilled

Outputs		
loc_v	-	VIP location
loc_{si}	-	Surveillance UAV i location
$vTrack_i$	-	Surveillance UAV i is tracking VIP

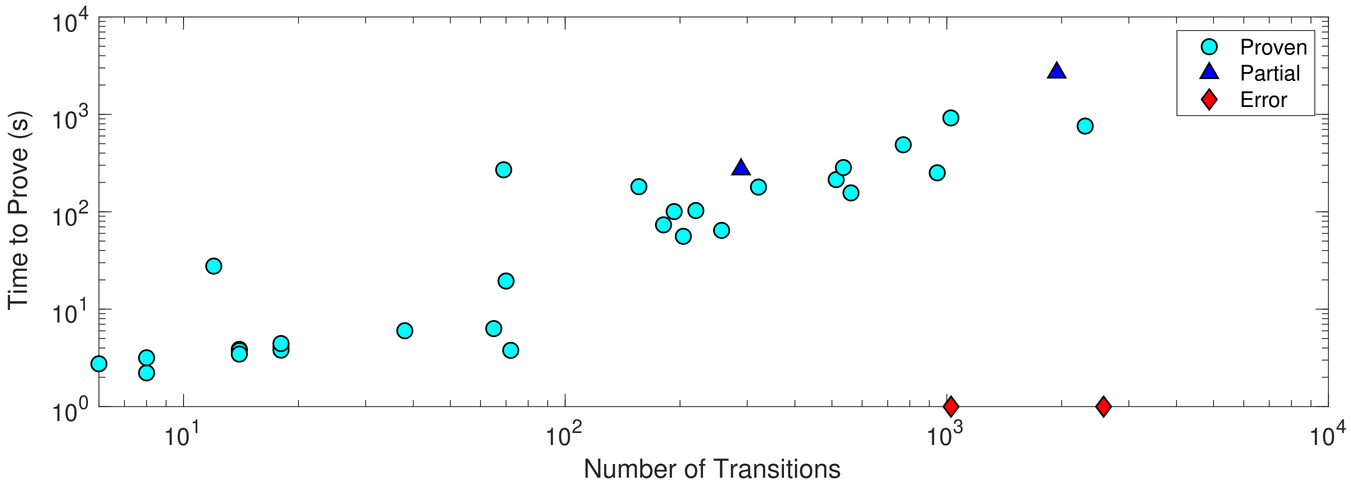
Case Study: Results

- Previously synthesized Python controller, interfaced it to OpenUxAS[11-12] to implement low-level UAV behaviors, simulated in OpenAMASE[13]
- Implementation in SPARK found undetected issue!
- Issue was part of specification for φ_t^e: $\square\neg(loc_e = i)$ for $i = \{1, 2\}$
- Specification should be written φ_t^e: $\square\bigcirc\neg(loc_e = i)$ for $i = \{1, 2\}$
- Reason: $\square\neg p$ not the same as $\square\bigcirc\neg p$
  - If environment chooses p for next time step, does not violate $\square\neg p$ in current time step
  - However, it necessarily violates $\square\neg p$ in next time step
  - φ_t^e should have terms of the form $\square\bigcirc\neg p$, not $\square\neg p$
  - Salty now produces warning in this case
- In Python implementation, result was states with no successors, leading to runtime errors after reaching those states

Results

SPARK Implementation of GR(1) Design: Summary

- Pulled 40 examples from GR(1) repos[2-6]: Salty, Slugs, Anzu, TuLiP, LTLMoP
- Plot shows CPU time for 33 examples with up to 4000 controller transitions
  - 7 examples with more transitions required too much memory (not plotted)
  - 2 examples had errors because of unusually large specifications
  - 2 examples only partially proven:
    - One with ~2000 transitions
    - One with Integer terms & arithmetic operators in specification



- Found the same type of specification error as UAV case study in one Anzu example

Summary

Lessons Learned

- "End-to-end" SPARK verification found issue due to $\Box\neg p$ vs. $\Box\bigcirc\neg p$
- `Case` statement more efficient proof-wise than `Map` for controller logic
  - SPARK `Maps` have complete axiomatization, adds verification conditions
    - During initialization, must prove each added key/input value not already contained
    - After initialization, must prove every possible transition covered
  - `Case` statements automatically imply these things
- `Case` statements have some issues:
  - Large/verbose
  - Underlying prover must reason about all alternatives at once
  - Still better than `Map` in our application, but maybe not for others
- Some examples had no inputs, vacuous precondition `Pre => True`
  - Took abnormally long time to verify
  - Clearly could be encoded more efficiently

Future Work

- Investigate liveness properties
  - SPARK hypothetically can solve required bounded model checking problem
  - Other option: annotate code & use external model checker

- Look at decomposition of `Move` procedure to handle larger examples

- Fix encoding of controllers with no inputs to speed up proof

- Investigate why example with Integer types & arithmetic operators takes longer than expected to prove in SPARK

References

Full Paper

For more details, see:

Humphrey, L. R., Hamil, J., Huguet, J.: "End-to-End Verification of Initial and Transition Properties of GR(1) Designs in SPARK," In: Int. Conf. Software Engineering and Formal Methods (SEFM), 2020.

References

1. 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)
2. Ehlers, R., Raman, V.: Slugs: Extensible GR(1) synthesis. In: Int. Conf. Computer Aided Verification (CAV), pp. 333–339. Springer (2016)
3. 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)
4. 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 (HSCC). pp. 313–314. ACM (2011)
5. 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)
6. 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)
7. Davis, J. A., Humphrey, L. R., Kingston, D. B.: When Human Intuition Fails: Using Formal Methods to Find an Error in the "Proof" of a Multi-agent Protocol. In: Int. Conf. Computer Aided Verification (CAV). pp. 366-375. Springer (2019)
8. Siegel, S. F.: What's wrong with on-the-fly partial order reduction. In: Int. Conf. Computer Aided Verification (CAV). pp. 478-495. Springer (2019)
9. J. W. McCormick & P. C. Chapin, *Building High Integrity Applications with SPARK*, Cambridge University Press, 2015.
10. AdaCore, "Introduction to SPARK," learn.adacore.com, 2018.
11. Rasmussen , S., Kingston D., Humphrey, L.: A Brief Introduction to Unmanned Systems Autonomy Services (UxAS). In: Int. Conf. Unmanned Aircraft Systems (ICUAS). IEEE (2018)
12. OpenUxAS on AFRL/RQ GitHub
13. OpenAMASE on AFRL/RQ GitHub